(4)

RADC-TR-90-203, Vol II (of three)
Final Technical Report
September 1990

DOS DESIGN/APPLICATION TOOLS
Software Top-Level Design Document

Honeywell Corp.

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

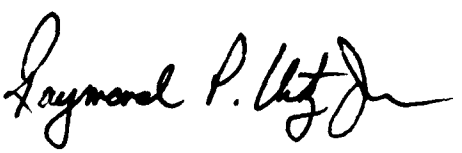
Rome Air Development Center
Air Force Systems Command
Griffiss Air Force Base, NY 13441-5700

This report has been reviewed by the RADC Public Affairs Division (PA) and is releasable to the National Technical Information Services (NTIS) At NTIS it will be releasable to the general public, including foreign nations.

RADC-TR-90-203, Vol II (of three) has been reviewed and is approved for publication.

APPROVED: 

THOMAS F. LAWRENCE
Project Engineer

APPROVED: 

RAYMOND P. URTZ, JR.
Technical Director
Directorate of Command & Control

FOR THE COMMANDER: 

IGOR G. PLONISCH
Directorate of Plans & Programs

If your address has changed or if you wish to be removed from the RADC mailing list, or if the addressee is no longer employed by your organization, please notify RADC (COTD) Griffiss AFB NY 13441-5700. This will assist us in maintaining a current mailing list.

Do not return copies of this report unless contractual obligations or notices on a specific document require that it be returned.

REPORT DOCUMENTATION PAGE

Form Approved
OPM No. 0704-0188

1. AGENCY USE ONLY (Leave Blank)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
		Final Dec 87 - Dec 89

4. TITLE AND SUBTITLE
DOS DESIGN/APPLICATION TOOLS Software Top-Level Design Document

5. FUNDING NUMBERS
C - F30602-87-C-0104
PE - 62702F
PR - 5581
TA - 21
WU - 78

6. AUTHOR(S)

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Honeywell Corporation
Sensor and System Development Center
1000 Boone Ave, North
Golden Valley MN 55427

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
Rome Air Development Center (COTD)
Griffiss AFB NY 13441-5700

10. SPONSORING/MONITORING AGENCY REPORT NUMBER
RADC-TR-90-203, Vol II (of three)

11. SUPPLEMENTARY NOTES
RADC Project Engineer: Thomas F. Lawrence/COTD/(315) 330-2158

12a. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution unlimited.

12b. DISTRIBUTION CODE

13. ABSTRACT (Maximum 200 words)
Developing applications for execution in a distributed processing environment is a difficult task. Such environments dominate Air Force C3I systems, which are necessarily distributed. In addition to being a physical necessity, distributed systems offer, relative to centralized processing systems, the potential for increased performance and fault tolerance. Realizing that potential is a key objective behind research in distributed systems technology.

The goal of this contract is to:
1. Define and demonstrate a framework for integrating development tools, AND
2. Define and construct tools that support the development of distributed applications.

A tool integration platform was designed and developed as a fundamental element of an integrated development framework. The RADC Distributed System Evaluation (DISE) Environment Tool Integration Platform integrates software development tools by automating and coordinating information exchange between tools, through use of the CRONUS distributed system and the ONTOS object oriented database management system. (Continued)

14. SUBJECT TERMS
Software Development Tools, Resource Allocation, Tool Integration, Reliability Analysis, Distributed System, Object Oriented DBMS

15. NUMBER OF PAGES
102

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	UL

Block 13 (Continued)

Two development tools were selected and implemented that illustrate the types of technology required to support distributed application development. The Allocator assists developers with determining efficient implementations for distributed applications. The Reliability Analyzer generates reliability measures for application components given a set of hardware reliabilities. The two tools have been integrated into the IP.

This report summarizes the contract's objectives and results.

PREFACE

This document is the Final Software Top Level Design Document for RADC contract F30602-87-C-0104, DOS Design Application Tools. The contractor is Honeywell's Corporate Systems Development Division. The document specifies the design of the Tool Integration Framework and Development Tools System. Requirements for the system are defined in the Draft System/Segment Specification document generated under this contract. The System presented in this document consists of three Computer Software Configuration Items (CSCIs), presented in separate parts, each of which is formated according to the Software Top Level Design Document Data Item Description.

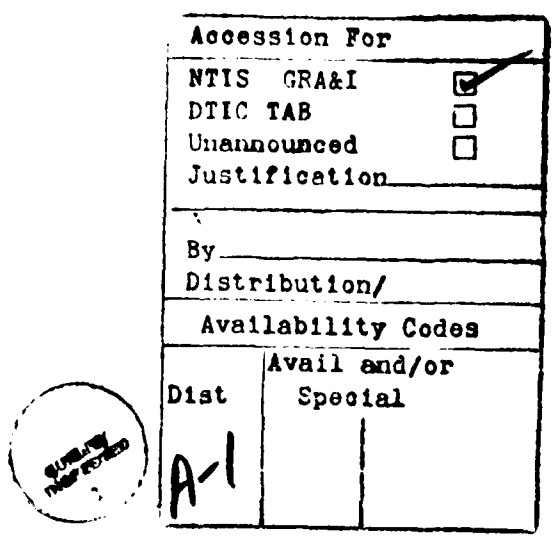


Table of Contents

List of Figures

PART I

TOOL INTEGRATION FRAMEWORK

1. Scope

1.1. Identification

This Software Top Level Design Document describes the top level design for the Computer Software Configuration Item (CSCI) identified as the Tool Integration Framework, CSCI Number 1, of the DISE Tool Integration Framework and Development Tools System. The requirements for the System appear in the Draft System/Segment Specification.

1.2. Purpose

An integrating framework provides methods and facilities that enhance the process of developing software. Integration assures easy transfer of information among tools, simplifies their use, facilitates consistency maintenance as applications are transformed from requirements to implementation and off-loads menial tasks from people to computers. Development tools in DISE are generally developed in isolation and therefore may not operate in a coordinated fashion. Furthermore, DISE is an evolving environment that will continue to acquire new, sophisticated development tools, such as instrumentation tools and performance analyzers, that need to be integrated into the tool base. The objective behind this CSCI is to develop a framework for integrating tools in RADC's DISE environment.

The purpose of the DISE Tool Integration Framework is to:

- support and encourage cooperation among (existing and new) independently developed programming tools by automating (or partially automating) data exchange and sharing among the tools, and
- capture and retain all information required to develop the application/system (information generated during system development).

Tool integration in this framework will be achieved using a project database (PDB) in which tools can store, and through which they share, information. Tools will interface with the PDB for their data input/output requirements. The PDB provides a canonical, well-defined and conceptually centralized database of tool input/output data entities. These entities are the various pieces that comprise the distributed application under development, such as source code modules and processing environment attributes. These entities are cast in an information model that characterizes the enterprise of developing distributed applications.

The PDB is a tool integration mechanism in the sense that it coordinates the interactions between tools; it simplifies the task of bringing a tool into the development environment and having it work with other tools in a coordinated fashion. While this approach is clearly not the only relevant notion of integration in a development environment, it plays an important role in a broader integrating framework and provides a platform upon which other integration mechanisms can be built.

In this framework the PDB will serve both as a repository of information and as the medium for information exchange among tools. This design specification focuses on the PDB and the nature of its support for the integration of development tools.

1.3. Introduction

This document presents a top level functional and abstract data type decomposition of the Tool Integration Framework CSCI and its single TLCSC (Top Level Computer Software Components). It also defines the interfaces between this CSCI and other CSCIs in this System, the Reliability Analysis Tool and the Allocation Tool. Included in the Notes (paragraph 6) are an external interfaces section, database schema design notes and a detailed requirements specification.

2. Referenced Documents

- *Draft System/Segment Specification*, DOS Design Application Tools, Contract No. F30602-87-C-0104, CDRL No. 3, 25 August, 1988, RADC/COTD, (Honeywell Inc., Corporate System Development Division).
- *Cronus User's Manual*, Cronus Release 1.2 BBN Laboratories, Inc., 15 January, 1988.
- *Tutorial Documents*, Cronus Release 1.2 BBN Laboratories, Inc., 15 January, 1988.

3. Requirements

This section presents the requirements for the Project Database (PDB). It begins with a discussion of the overall architecture of the PDB as it relates to DISE, Cronus and development tools. Section 3.4 presents some additional details on the current functional decomposition of the PDB. Detailed designs of the main PDB components are provided in section 3.5.

3.1. CSCI Architecture

This section shows how the PDB fits in DISE and presents a top level architecture of the PDB itself.

3.1.1. DISE Tool Integration Framework

The framework architecture as it relates to DISE is depicted in Figure 1. The PDB is encapsulated in a Cronus object manager that is located on a computing platform in the distributed computing environment. Each platform in the network hosts both a local or constituent operating system (COS) and the Cronus distributed operating system (DOS). Software development tools execute as Cronus applications, accessing the PDB through communication constructs provided by the DOS. This architecture

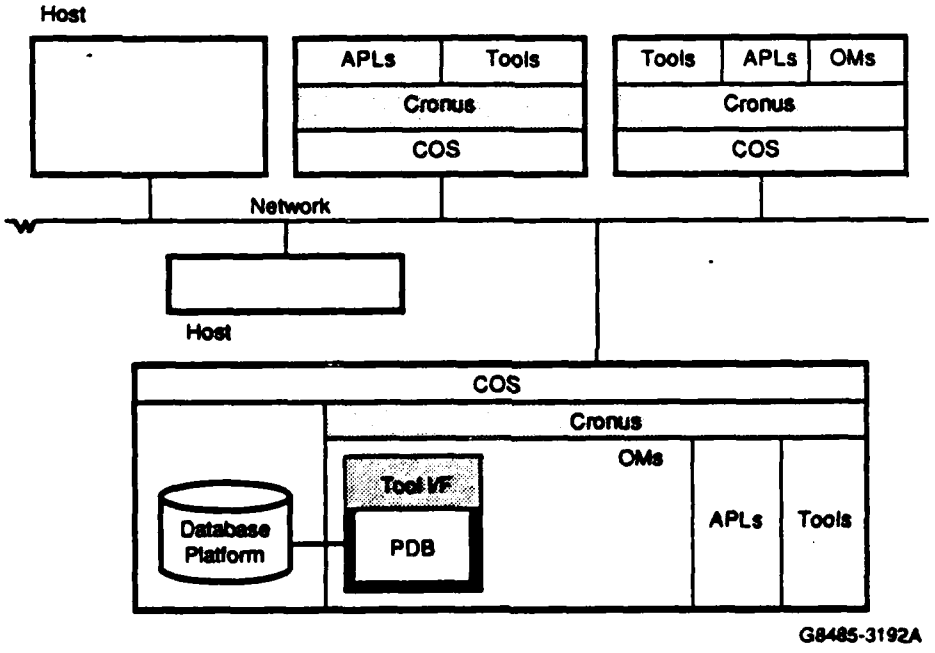


Figure 1. DISE Tool Integration Architecture

supports both distributed development, where developers anu their tools may be dispersed across the network, as well as tools that operate in a distributed manner, such as performance measurement tools and distributed debuggers.

This architecture supports distributed application developers in two important ways. First, by virtue of the DOS's services, it buffers them from problems related to distribution that arise when using a network of computing platforms for development. Second, it offers them a potentially diverse set of development tools that work in a coordinated fashion with respect to the program under development.

Tool developers also benefit from this architecture. The DOS hides problems related to platform distribution. This allows them, when implementing a tool, to selectively utilize the different processing resources available in the distributed environment without concern for compatibility with other platforms. More importantly, the architecture allows tool developers to focus on the tool's function and the single project information repository made readily accessible in the network through DOS services. A tool developer therefore is not forced to understand, implement and coordinate the interactions it has with each of the (potentially large set of) other tools in the environment, interactions which would necessitate understanding the input/output data formats of each other tool, and its location and host computing platform. The architecture shown in Figure 1 benefits both distributed application developers as well as tool developers by building upon the DOS services and encapsulating the PDB in an object manager.

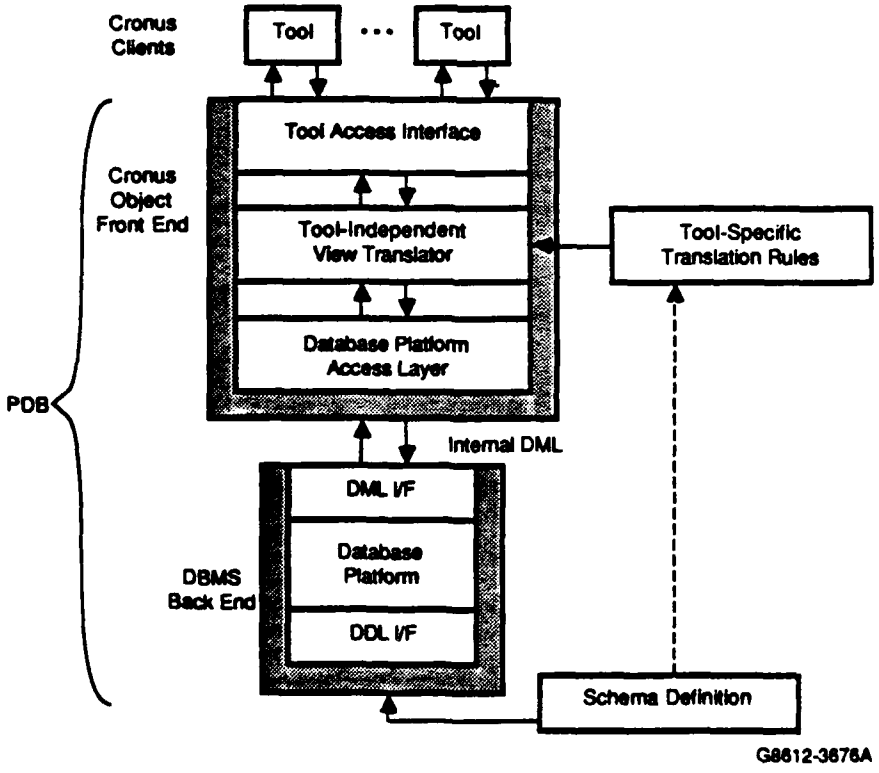


Figure 2. Project Database Architecture

3.1.2. Project Database Architecture

Figure 2 depicts the CSCI top-level architecture for the Project Database. The PDB is a Cronus object manager that acts as a front end to an internal database platform. The PDB's Tool Access Interface provides a set of operations that tools can invoke to access data in the database. Tools make calls on this interface for their data needs, for example, to retrieve particular processin environment attributes.

The Tool Access Interface operations invoke the Tool-Independent View Translator. This translator applies tool-specific translation rules that define a mapping between the external (tool) and internal (PDB) views of the data. The rules translate external data references issued by a tool into the appropriate data manipulation language (DML) code fragments that are developed in accordance with the PDB schema definition. The DML fragments applied by the View Translator on behalf of a tool provide access to the desired data in the underlying database platform. Access occurs through the Database Platform Access Layer, a programmatic DML interface provided by the database platform.

Some of the advantages of encapsulating a database platform within a Cronus object are listed below:

- All issues relating to distribution (message routing, location transparency, naming, etc.) are handled by the Cronus distributed O/S. The database can, therefore, support distributed tools and tools that run at remote sites.
- By encapsulating all database implementation dependencies, a consistent external interface can be provided to the tools. The different tools in the supported toolset will interact with the database through this tool access interface only and see the PDB as just another Cronus object. As all Cronus applications interact with Cronus managers/objects, this architecture allows the different tools running as Cronus applications to access data in the database via a similar programming interface.
- The presence of such a consistent programmatic interface to the PDB facilitates integration of new tools into this framework.
- Cronus internals will not be modified. Our approach will entail creation of a specialized Cronus Manager to contain the PDB, but will require no changes to the Cronus kernel or any of the existing tools/managers.

Schema definition and modification are effected on the database platform directly using the platform's native data definition language (DDL) and associated facilities. These changes are made by the project/database administrator, and not by a variety of tools, so providing remote access to these facilities via Cronus is not seen as particularly beneficial. Schema maintenance requires detailed knowledge of the database platform's DDL facilities and the information model, and are typically made relatively infrequently.

3.2. Functional Allocation

The requirements for this CSCI are defined in the System/Segment Specification document identified above in section 2. In particular, the requirements for the Tool Integration Framework specified in section 3.1.6.1 of that document identify requirements for four functions: information model and database schema definition, tool access capability, type definition and schema extension capability and user interface capabilities. All these functions are allocated to the single Database TLCSC.

3.3. Memory and Processing Time Allocation

Not specified.

3.4. Functional Decomposition

Section 3.1.2 presented the top-level design architecture of the PDB. This section presents a slightly more detailed picture of the current approach to implementing that design. The first aspect to note is the selection of a particular database that will serve as the underlying database platform. The selected product is Vbase[1], a commercial object-oriented database available for UNIX[2] environments. It provides a data manipulation language (DML) called COP, which is an extension of the C language, and a data definition language (DDL) called TDL for defining object types.

Figure 3 shows the functional decomposition of the PDB into a set of processes. Application developers apply tools, each a separate process, through tool-specific interfaces. As discussed above, these development tools interface with the PDB object for their data requirements using the Cronus Interprocess Communication (IPC) facilities. The View Translator in this design is broken out as a separate process. The PDB object interfaces to the View Translator using local (UNIX) IPC facilities. The View Translator is written in COP and accesses Vbase through its programmatic interface. The View Translator applies tool-specific translation rules, also written in COP, to access the database in response to a tool's request. Finally, database maintenance and storage management are handled transparently within Vbase.

This architecture provides a number of benefits in addition to those identified in section 3.1. In particular, the PDB object is not affected by the addition of new translation rules, implying at least that the PDB object does not have to be recompiled/restarted with each such addition. In fact the intermediate View Translator process allows the PDB object to be independent of Vbase, in effect shielding the PDB object from the specifics of the underlying database platform. Finally, this design may possibly provide some flexibility with respect to supporting multiple concurrent tool accesses on the database.

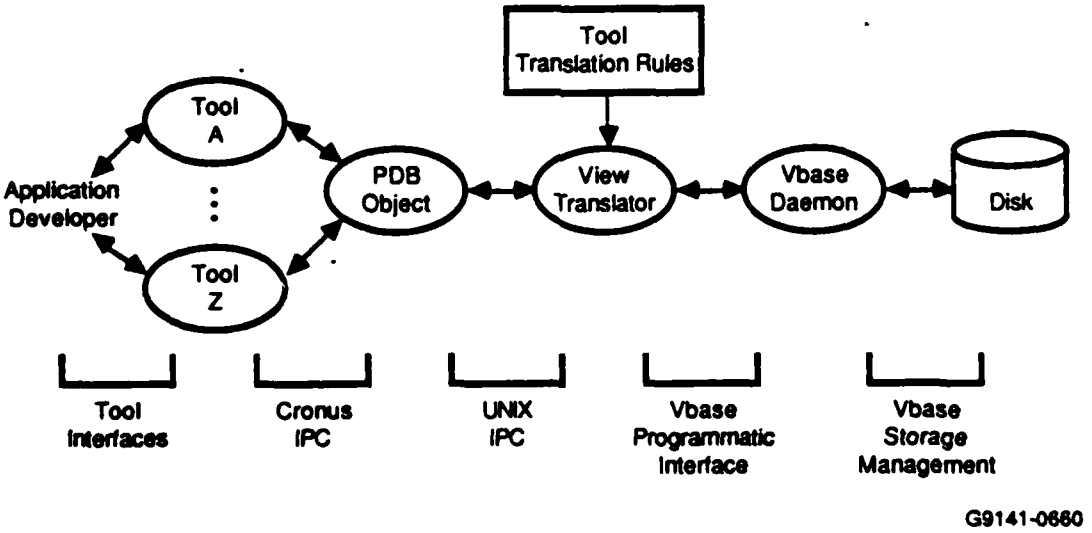


Figure 3. Project Database Functional Decomposition

[1]Vbase is a trademark of Ontologic, Inc.

[2]UNIX is a trademark of AT&T Bell Laboratories.

3.5. Top Level Design

3.5.1. Database TLCSC

3.5.1.1. Identification

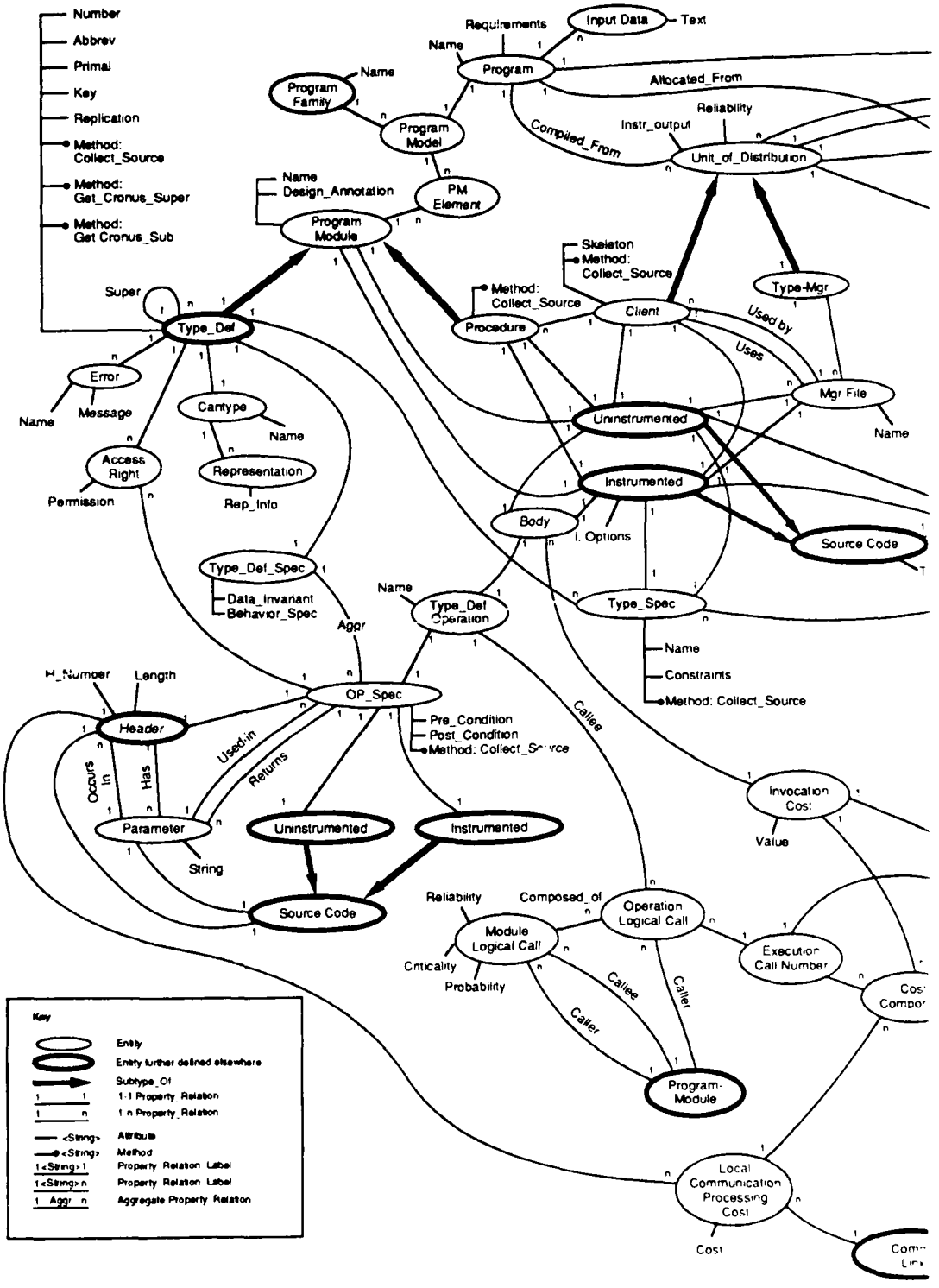
This design specification focuses on the four primary features of the database: the schema, the translation rules, the tool access interface and tool/PDB coupling. Section 3.5.1.2 below presents the database schema that captures the information needed by application development tools. Section 3.5.1.3 describes the translation rules that provide a means for defining a tool's particular view of the database schema. The tool access operations exported by the PDB object for tools to use are presented in section 3.5.1.4. The two methods of integrating tools with the PDB are discussed in section 3.5.1.6.

3.5.1.2. Database Schema

3.5.1.2.1. Approach

A schema characterizes an enterprise as the schema designers perceive that enterprise. These are the points of view present in the PDB schema:

- *Software engineering within the Cronus world.* We modeled neither strictly from the Cronus point of view nor from a vanilla software engineering point of view, but from the point of view of the software engineering enterprise carried out in DISE/Cronus. More Cronus-specific detail could be included in this schema, depending on user requirements; i.e., the queries practicing developers and actual tools run against the database.
- *Different data models among tools.* We do not require that all tools have the same model of their data. Instead we provide a schema that models a neutral view and translators to provide each tool with a tailored view.
- *Technical rather than project management point of view.* There is a great deal of detail about the elements of the program under development and enough information about, say, versions or configurations to "make" a program, but no purely management information such as sign-offs, deadlines, deliverables and the like. However, there is nothing in the data modeling technique that prevents the extension of the schema with project management objects, operations and relations.
- *Program development rather than running programs.* The project DB is not for storing or controlling results a distributed application program produces during operation. It can, however, store some program output (such as the results of instrumentation runs) that is used in program development.
- *Design point of view.* The design point of view is the one people have when they do development. We support that point of view, providing methods for the routine work of gathering up some elements of the program under development to present to tool such as a compiler in the proper syntax. We selected appropriate methods by examining tool inputs and outputs.
- *Tool level of abstraction.* The schema models at a different level of abstraction than the view designers have, because tools see the world at a different level of granularity than people. For example, the schema models a type operation's specification as: name, input parameters, return values, precondition, postcondition. A designer is perfectly capable of recognizing and using these parts separately, but is most likely to think of and refer to them all as "the specification." Tools, on the other hand, sometimes use only parts of "the specification;" for example, *genmgr* needs a type operation's name, but not the rest of its specification.



G8603-3434C

Figure 4. Pr

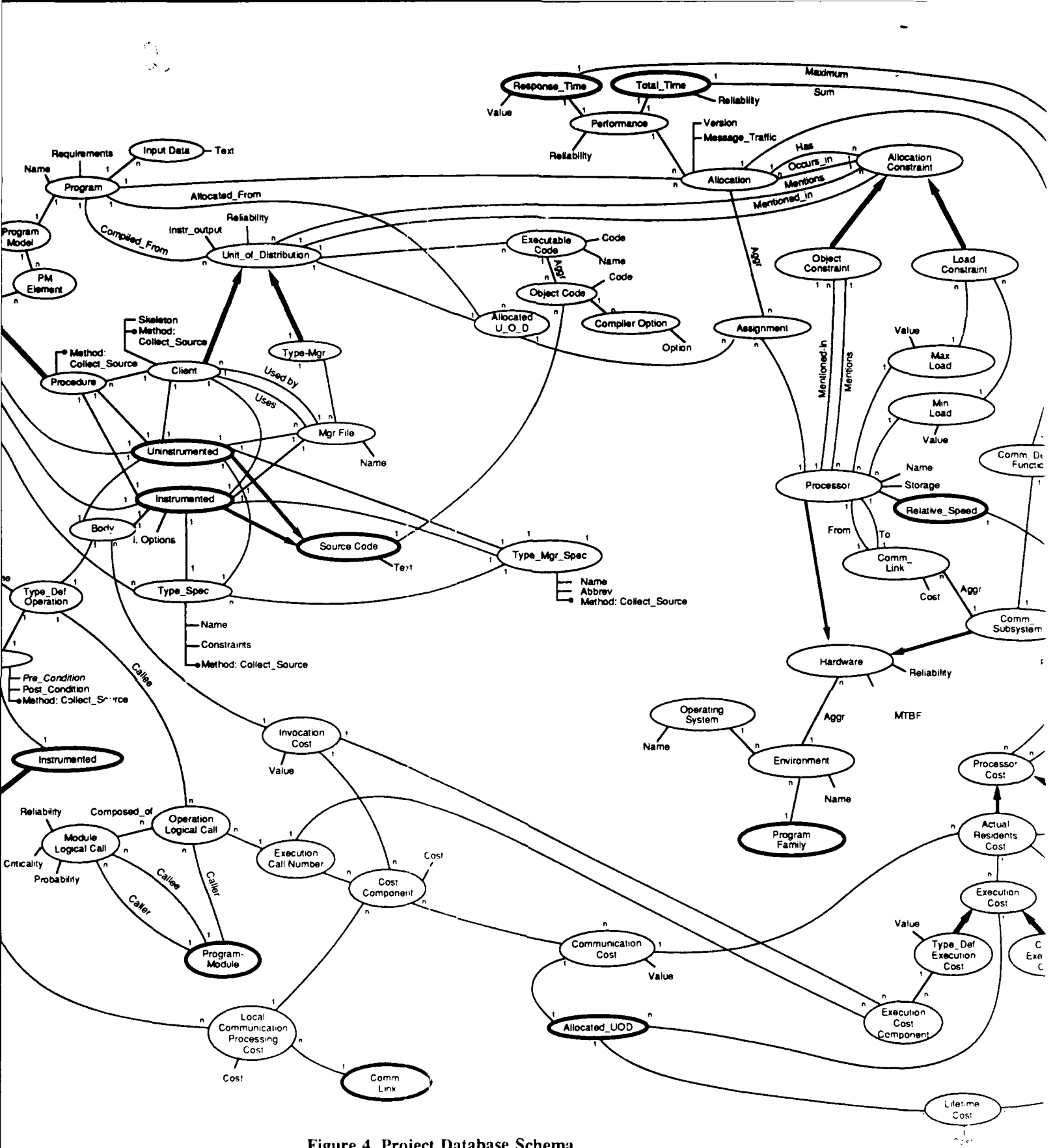
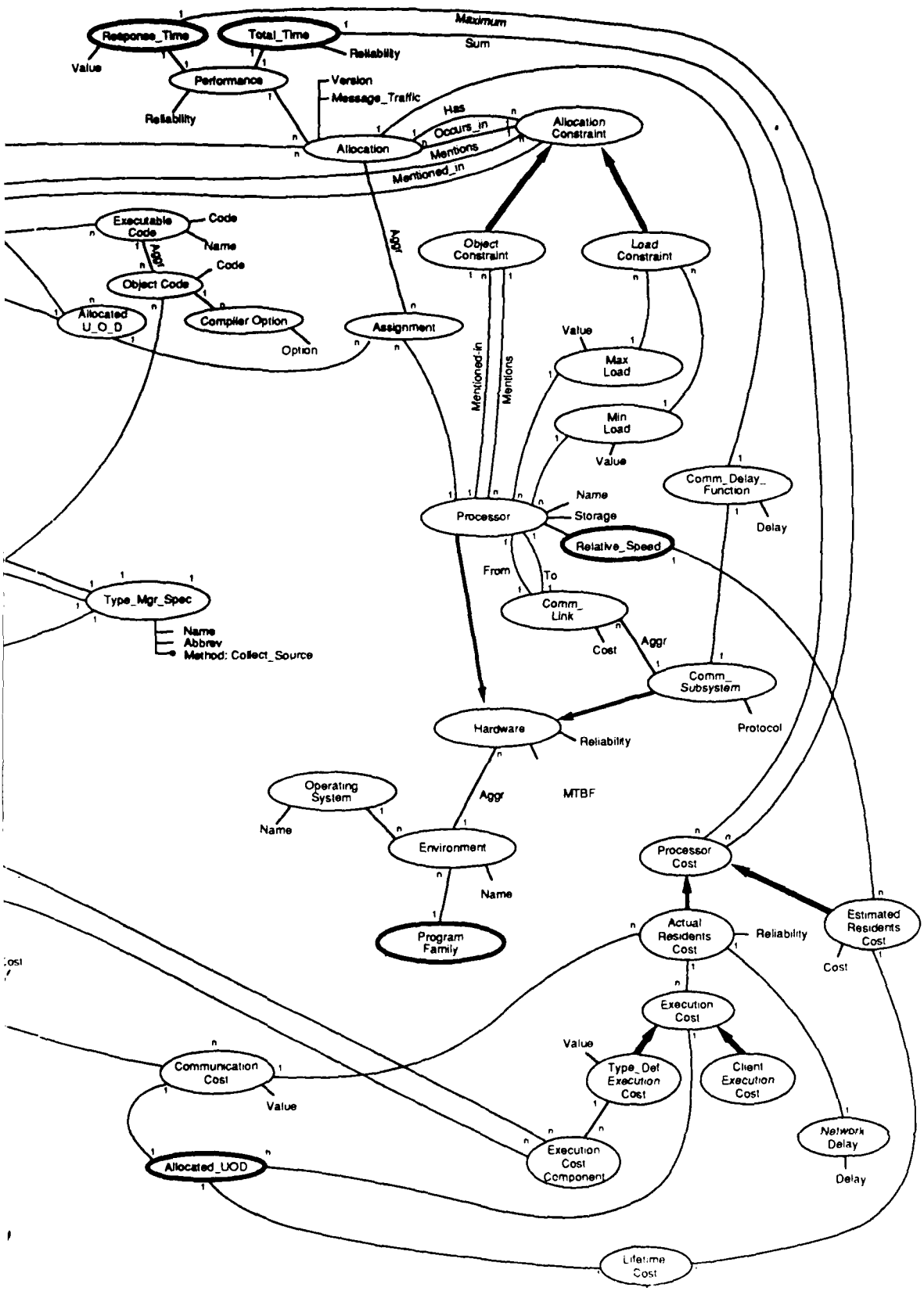


Figure 4. Project Database Schema



t Database Schema

This schema models data for these tools: Allocation tool, Reliability Analysis tool, text editor, Cronus *definetype*, Cronus *genmgr*, a compiler and instrumentation. Data can also be accessed by *ad hoc* query using the PDB's native query language.

Figure 4 shows the entities, attributes and relationships to be represented in the database schema. Ovals represent the entities, arcs between the ovals represent relationships between entities and attributes are represented by line segments with labels. The diagram includes additional information about hierarchical relationships between entities, represented by a bold arrow with the subtype at the head of the arrow. For example, Assignment is an entity participating in three relationships, one each with Allocation, Processor, and Allocated Unit_of_Distribution. Assignment has no attributes associated with it. Type_Def is a subtype of Program_Module. This means that in addition to the other attributes and relationships that are directly represented for Type_Def, it also has associated with it the attributes of Program_Module relationships to two subtypes of Source Code, Implemented and Unimplemented. The degree of a relationship is represented by annotations to the relationship arcs. A 1-n relationship is represented as an arc with a "1" at one end and an "n" at the other end. The diagram is a graphical representation of the entities and relationships that will be represented as objects and properties in the formal database schema discussed below.

3.5.1.2.2. Schema Definition

The formal database schema is a collection of object and method definitions stated in terms of the data definition language, TDL, used by Vbase. Each object class is defined as having a name and properties and a supertype. The values of the properties may be other objects defined in the schema or standard objects representing base types, e.g., integer, string. Objects may have associated methods. The methods are expressed using COP, a data manipulation language provided with Vbase.

The Appendix contains the list of types in the formal schema. It represents the information characterized by Figure 4. Most entities in Figure 4 are represented as objects in the formal schema. The relationships between two objects A and B are represented as properties of an object A whose value is of type B. The property names correspond to labels on the relationship arcs when those arcs are labeled. Typically only one direction of a relationship is represented. The other direction can be derived from the data in the database. In cases where the 1-n direction is explicitly represented it is captured in the Vbase schema as a property whose value is a distributed set of objects of some type. The properties of an object also represent the attributes of the entity in the drawing. The names of those properties correspond to the names of the attributes being represented. Subtype information from the drawing is presented in the supertype declaration of the object definition. Objects with no other supertype in the schema are represented as objects whose supertype is Type, a predefined type in Vbase. Type provides useful facilities for object creation and manipulation that are inherited by all objects in this schema.

3.5.1.3. View Translation

Tools use the database as a repository of information and as a medium for information exchange. Database operations allow the different tools to access data in the database. A tool access interface implemented at the Cronus object interface level provides DML calls that are used by the tools. The calls supported by this interface are *retrieve, update, copy, exists, create* and *delete*. These calls are processed using the invoking tool's translation rules and translated into data access calls in the PDB's native DML. The next two subsections describe our approach to view translation and provide some example translation rules.

3.5.1.3.1. Translation Rules

A database schema is commonly thought of as defining a "union of user views" of the data in the database. Typically, users want shared access to overlapping data. To provide shared access it is necessary

to define a schema from which multiple user data needs can be derived because the schema itself will not match the data needs of every individual user. In this environment the users of the database include software developers and the application development tools. Since the database is intended to support future tools as well as existing tools, and since there may be conflicts in the data needs of existing tools, it will not be possible to define a schema which is tailored to the data needs of all existing and future users of the database. The solution is to define a schema which supports user requirements by allowing required data to be derived from information stored in the database and a uniform mechanism for deriving the data needed by individual users (i.e., development tools and software developers).

Another reason to provide a uniform mechanism for deriving data is the inflexibility of the Cronus object interface. Any change to this interface requires the PDB object and manager to be brought down and recompiled. To accommodate the data needs of new tools in this evolving environment, tools in the system access the PDB via the same interface but have their own *views* of the data in the database by *deriving* their information from the database via tool-specific translation rules.

The uniform mechanism in this approach is a *view translation* module. The view translator requires each tool to specify a set of view translation rules. View translation rules specify the correspondence between tool's data structures and elements of the database schema. These translation rules define a view on the database tailored to the associated tool. The tool developer must specify the rules and must invoke (indirectly, through the tool access interface) the view translator for access to database items. The view translator will accept the request, translate it to an appropriate database expression, access the database, and return the results to the tool in terms of the tool's local data structures.

The general form of a translation rule is:

<Tool_name, Op_name, Rule_selector> ==> <database expression>

The <Tool_name, Op_name, Rule_selector> tuple uniquely identifies a particular rule. Tool_name specifies the invoking tool, and Op_name is the name of the operation being performed. Since a tool may use an operation on a variety of different database entities, a rule_selector parameter is introduced to identify the rule corresponding to a particular database entity for a given tool and operation. For example, for the *update* operation the Rule_selector might simply be the name of the tool variable to be modified, as it would uniquely identify the database expression to be applied in order to modify the corresponding database entity.

The right hand side of the rule is a database expression written in the internal DML. Depending upon the operation being performed and the type of entity involved, the expression may require various input and output values which correspond to tool-specific variables. For example, a rule involving the *update* operation would require one or more input values from the tool. An *retrieve* operation might require a single output parameter to hold the value retrieved. These tool-specific variables can be any type defined or used in the tool, for example, an unstructured variable such as the descriptor of a record field or an array cell. The types in the database expression must be compatible with the types of the corresponding tool variables. For example, the database expression cannot return a set of values if the tool variable is defined to be a single value. If the tool requires multiple values from the database, the tool must support the restructuring of those values in a form which can be manipulated locally, by requesting the values from the translator at the appropriate level of granularity. The translation rules provide database access using only those capabilities defined by the database DML.

The internal DML for the PDB is the COP DML of Vbase. Rules may be specific to a particular PDB, necessitating a fourth parameter (the PDB name) in the left hand side of a rule. Translation rules then take the following form:

<Tool_name, Op_name, Rule_selector, OPTIONAL DB_name> ==> (COP code)

The COP code body, or the code to be executed to actually access the required data item in the database, can be stored in a separate tool-specific file. This file, containing COP code fragments, could be compiled and linked with the View Translator and applied through the database's programmatic interface. Or, the code for the translation rules relevant to a tool can be stored in the database in a *translation_rule* object, instantiated on a per-tool basis, and applied from within the database itself.

The View Translator applies a translation rule as a result of a Tool Access Interface call issued by a tool. As discussed above, the COP code fragment thus obtained typically requires from the tool some input and output parameters. The View Translator executes the COP code fragment in the context of these parameter values to carry out the desired operation on the database. The View Translator may provide standard facilities for opening and closing the database and any other functions which must be performed for all translations.

The tool developer is responsible for writing the tool's translation rules and for invoking the appropriate call on the Tool Access Interface (which in turn invokes the View Translator). Several points must be observed by the tool developer to ensure successful application of the translation rules:

- Database reference must be correctly relativized.
- Results of database expressions must be of the correct type and granularity for the tool variable.
- Errors should be caught and the appropriate exceptions raised.

References to database objects must be correctly relativized. For example, if the tool assumes that a variable, say Type_Definition, refers to a particular type definition (e.g., a type included in the Type manager for Unit_Of_Distribution UID), then the translation rule database expression must reflect that relativized reference. It would be an error to simply translate the tool variable Type_Definition as the database object Type_Def. The correct translation would be something like the COP fragment:

```
Type_Definition =

        Input: (Unit_Of_Distribution, Type_Manager, Type_Manager_Spec,
                Type_Def)
        Output:  result
        DB_name: Project_db
        Code:

        ud_id = Entity$NameLookup("Unit_Of_Dist$UID")
        result = ud_id$Type_Manager_Of$Spec_Of$Type_Def_Of$Type_Def_Name
```

The extended path name specifying the value of result mentions the property names of the objects and the actual objects mentioned in the import list. The final result expression must be of type String to be compatible with the type of the tool variable Type_Definition. A common error would be to use the expression below instead of the correct expression.

```
result =  ud_id$Type_Manager_Of$Spec_Of$Type_Def_Of
```

The value of this expression would be a database internal pointer to the database object Type_Def and not a string value compatible with the tool variable Type_Definition.

The approach to tool access of the database through a view translation mechanism will provide modularity to both database maintenance and tool development. Changes in the database schema may result in changes in the view translation rules for a tool but not in changes to the tool itself. Different database management systems may also be substituted without requiring change to the tool's internal code; all changes would be localized to the view translator mechanism.

3.5.1.3.2. Example Translation Rules

Here are some sample rules for translating tool requests for data retrieval/update into database accesses. They are specific to the schema presented in subsection 3.5.2.1.2 and are written as COP (C Object Processor) language fragments. Translation rules are not needed for all data structures used by the tool, only for those that the tool (or tool envelope) mentions in database calls. The implemented translation tools will use COP to reference the PDB; people may use Vbase SQL for *ad hoc* queries.

Examples from the Allocation Tool

TRANSLATION RULE for Allocation Tool Update:

```
Tool_Name: Allocation_Tool
Op_Name: update
Rule_selector: New_Assignment
OPTIONAL DB_Name: PDB

==>

Input parameters: Processor.name, Unit-of-Distribution.identifier
Output variables: Assignment_Info.assign
Code:

    obj Assignment the_assignment;
    obj Allocation the_allocation;
    obj Unit-of-Distribution the_uod;
    obj Processor the_processor;

    {
     the_allocation = $Entity$NameLookup ("Allocation$current");
     the_uod = $Entity$NameLookup ("Unit-of-Distribution$Unit-of-Distribution.identifier");
     the_processor = $Entity$NameLookup ("Processor$Processor.name");
     the_assignment = Assignment$[
                        Platform: the_processor
                        Software: the_uod
                        occurs-in: the_allocation ]
    }
```

TRANSLATION RULE for Allocation Tool retrieve:

```
Tool_Name: Allocation_Tool
Op_Name: retrieve
Rule_selector: Assignment_performance
OPTIONAL DB_Name: PDB

==>

Input parameters: none
Output variables: Assignment_Info.performance
Code:
```

```
obj Allocation the_allocation;
Float result;

{
the_allocation  = $UnorderedDictionary$GetElement (AllocDictionary, "current");
result = the_allocation.cost;
}
```

Examples from Cronus definetype

We retrieve the following values from the PDB to give to Cronus *definetype*. Each retrieval could have been written as a separate translation rule. The application doing the retrieval is the envelope for *definetype*.

TRANSLATION RULE for *definetype* retrieve:

```
Tool_Name: "definetype"
Op_Name: retrieve
Rule_selector: which_type
OPTIONAL DB_Name: PDB

==>

Input parameters: which_type
Output variables: name_result, number_result, abbrev_result,
                  subtype-of_result, key_result, annotation_result,
                  cantype_result
Code:

    obj Type-def the_type_def;
    String name_result, number_result, abbrev_result, subtype-of_result,
           key_result, annotation_result, cantype_result;

    {
    the_type_def  = $UnorderedDictionary$GetElement (TypeDefDictionary , which_type);
    name_result = the_type_def$name;
    number_result =  the_type_def$number;
    abbrev_result =  the_type_def$abbrev;
    subtype-of_result  = the_type_def$supertype$name;
    key_result  = the_type_def$key;
    annotation_result = the_type_def$design-annotation;
    cantype_result = the_type_def$cantype_representation$rep_info;
    }
```

Examples from Cronus genmgr

We retrieve the following value from the PDB to be part of the type manager specification given to Cronus *genmgr*. The application doing the retrieval is the envelope for *genmgr*; the OUTPUT VARIABLES used are those found in *genmgr*.

TRANSLATION RULE for *genmgri* retrieve:

```
Tool_Name: "genmgr"
```

```
Op_Name: retrieve
Rule_selector: which_mgr
OPTIONAL DB_Name: PDB

==>

Input parameters: which_mgr
Output variables: abbrev
/* We assume an operation is unimplemented when its uninstrumented source code slot is empty.
This is indicated when has-value returns 0.*/

    Code:

        obj Type-Manager-Spec the_type_manager;
        the_type_manager = Entity$NameLookup("Type-Manager-Spec$which_mgr");
        {
        abbrev = the_type_mgr$abbrev;
        }
```

Example of Editor Input

This is an example of gathering up the source code extension of the PDB client object for a particular Cronus client under development. (Source code is collected and stored in an object's Source slot by a method called Collect-source).

TRANSLATION RULE for Editor retrieve:

```
Tool_Name: "Editor_X"
Op_Name: retrieve
Rule_selector: which_client
OPTIONAL DB_Name: PDB

==>

Input parameters: which_client;
Output variables: file;
Code:

    obj Client the_client;
    File editor-file;
    {
    the_client = $Entity$NameLookup("Client$which_client");
    the_client$Collect-source; /*method invocation*/
    editor_file = Client$Source;
    }
```

3.5.1.4. Tool Access Interface: PDB Object Operations

Tools access the PDB through its Tool Access Interface. There are six data access calls supported by the Tool Access Interface in the Cronus PDB object: *retrieve, update, copy, exists, create* and *delete.* These calls allow the tool to retrieve and manipulate application information retained in the PDB as

dictated by the needs of the tool. The calls have the natural semantics:

retrieve: Obtain a database entity to be used by the tool.

update: Modify attribute values of a database entity.

copy: Copy one database entity to another database entity.

exists: TRUE if entity exists in the PDB.

create: Create an instance of an entity type.

delete: Delete an instance of an entity.

The general syntax of a call is:

Op_name(Tool_name, Rule_selector, [OPTIONAL DB_name], <List of Tool variables>)

Example calls would look like these:

retrieve ("Allocation_Tool", Type_Def, PDB, Type_Definition, result)

update ("Allocation_Tool", Type_Def_Name, PDB, Type_Definition.name, "George")

The calls made by tools to invoke these primitives and the required parameters for each of these calls are given below.

Data retrieval:

retrieve (tool_name, Rule_selector, [PDB], variable_of_interest, result)
RETURNS call_status

```
BEGIN
locate_db(PDB) => pdb;
get_translation_rule(tool_name, retrieve, Rule_selector, pdb) => COP_code;
invoke(pdb, COP_code, variable_of_interest, result) => call_status;
return(call_status);
END
```

Raises TYPE_MISMATCH if *result* is of incorrect type
Raises NOT_FOUND if translation rule for *Rule_selector* is not found
Raises NO_PERM if permission is denied

Data updates (inserts & deletes):

update(tool_name, Rule_selector, [PDB], variable, new_value)
RETURNS call_status

```
BEGIN
locate_db(PDB) => pdb;
get_translation_rule(tool_name, update, Rule_selector, pdb) => COP_code;
invoke(pdb, COP_code, variable, new_value) => call_status;
return(call_status);
END
```

Raises TYPE_MISMATCH if *new_value* is of incorrect type
Raises NOT_FOUND if translation rule for *Rule_selector* is not found
Raises NO_PERM if permission is denied

Create instances:

create(tool_name, Rule_selector, [PDB], object_type, instance_name)
RETURNS call_status

```
BEGIN
locate_db(PDB) => pdb;
get_translation_rule(tool_name, create, Rule_selector, pdb) => COP_code;
invoke(pdb, COP_code, object_type, instance_name) => call_status;
return(call_status);
END
```

Raises NOT_FOUND if translation rule for *Rule_selector* is not found
Raises NO_PERM if permission is denied

Delete instances:

delete(tool_name, Rule_selector, [PDB], instance_name)
RETURNS call_status

```
BEGIN
locate_db(PDB) => pdb;
get_translation_rule(tool_name, delete, Rule_selector, pdb) => COP_code;
invoke(pdb, COP_code, instance_name) => call_status;
return(call_status);
END
```

Raises NOT_FOUND if translation rule for *Rule_selector* is not found
Raises NO_PERM if permission is denied

Copy data:

copy(tool_name, Rule_selector, [PDB], from_var, to_var)
RETURNS call_status

```
BEGIN
locate_db(PDB) => pdb;
get_translation_rule(tool_name, copy, Rule_selector, pdb) => COP_code;
invoke(pdb, COP_code, from_var, to_var) => call_status;
return(call_status);
END
```

Raises NOT_FOUND if translation rule for *Rule_selector* is not found
Raises TYPE_MISMATCH if *from_variable* and *to_variable* are not of the same type
Raises NO_PERM if permission is denied

Check if instance exists:

exists(tool_name, Rule_selector, [PDB], instance_name, result)
RETURNS call_status

```
BEGIN
locate_db(PDB) => pdb;
get_translation_rule(tool_name, exists, Rule_selector, pdb) => COP_code;
invoke(pdb, COP_code, instance_name, result) => call_status;
return(call_status);
```

END

Raises NOT_FOUND if translation rule for *Rule_selector* is not found
Raises NO_PERM if permission is denied

3.5.1.5. Tool Integration: Tool/PDB Coupling

This section discusses the manner in which tools are coupled to the PDB. Two types of coupling are supported in this framework: direct and indirect. Directly-coupled tools are those tools that are developed in the context of the PDB. The tool developer in this case is aware of the PDB and its associated information model, and develops the tool accordingly. Indirectly-coupled tools are those tools that are constructed outside the context of the PDB and its information model. Such tools include both existing tools as well as tools that are brought in from foreign environments. Both types of tools are supported in this integration framework.

3.5.1.5.1. Directly-Coupled Tools

Directly-coupled tools interface to the PDB through two separate logical channels. As shown in Figure 5, the body of a directly-coupled tool interfaces with the PDB object through the Tool Access Interface. It is through this interface that the tool obtains data from and stores data into the information repository. The execution of one of these calls results in an appropriate translation rule being applied on the database to access data. The translation rules themselves, on the other hand, are more tightly coupled to the PDB than the tool's body in the sense that they are incorporated within the underlying database platform directly as either applications or internal methods. The translation rules, while interfacing to the PDB through a different channel than the tool's body, are nevertheless specific to that tool and are developed hand-in-hand with the development of the body.

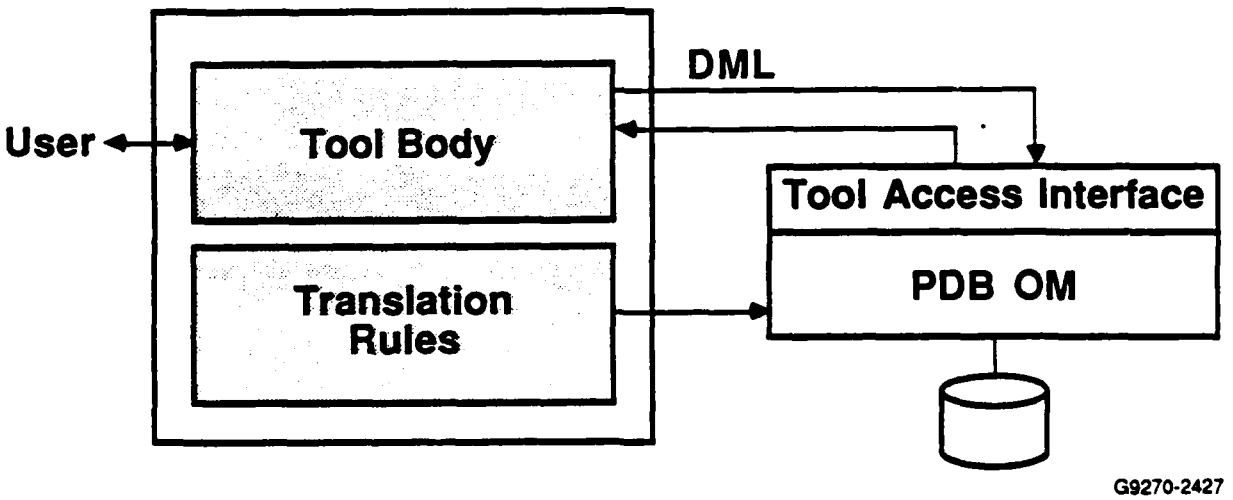


Figure 5. Directly-Coupled Tools

3.5.1.5.2. Indirectly-Coupled Tools

Indirectly-coupled tools also interface to the PDB through two different logical channels. As diagramed in Figure 6, these tools are integrated by encapsulating the entire tool in a *tool envelope.* This envelope performs pre- and post-invocation processing to convert between the encapsulated tool's input/output data and the corresponding data in the PDB. The envelope invokes the necessary operations at the PDB's Tool Access Interface. Indirectly-coupled tools also have an associated set of translation rules that work in just the same way as they do for directly coupled tools. However, they are developed in conjunction with the envelope, not the tool itself, and are still coupled to the PDB directly. This approach supports integration of existing and foreign tools by augmenting the tool's interface rather than requiring changes the the tool's internals.

3.6. Adaptation Data

The Tool Integration CSCI executes on a Sun workstation as a Cronus object. The host workstation environment must therefore provide Sun workstations running UNIX and the Cronus distributed operating system.

6. Notes

6.1. Interfaces

6.1.1. PDB Interface to Cronus

The PDB is encapsulated in a Cronus object and therefore a Cronus system manager must be created, compiled and instantiated to manage it. This manager will support one generic operation to create an instance of the PDB object. The Cronus operations specific to the PDB are invoked by the tools and

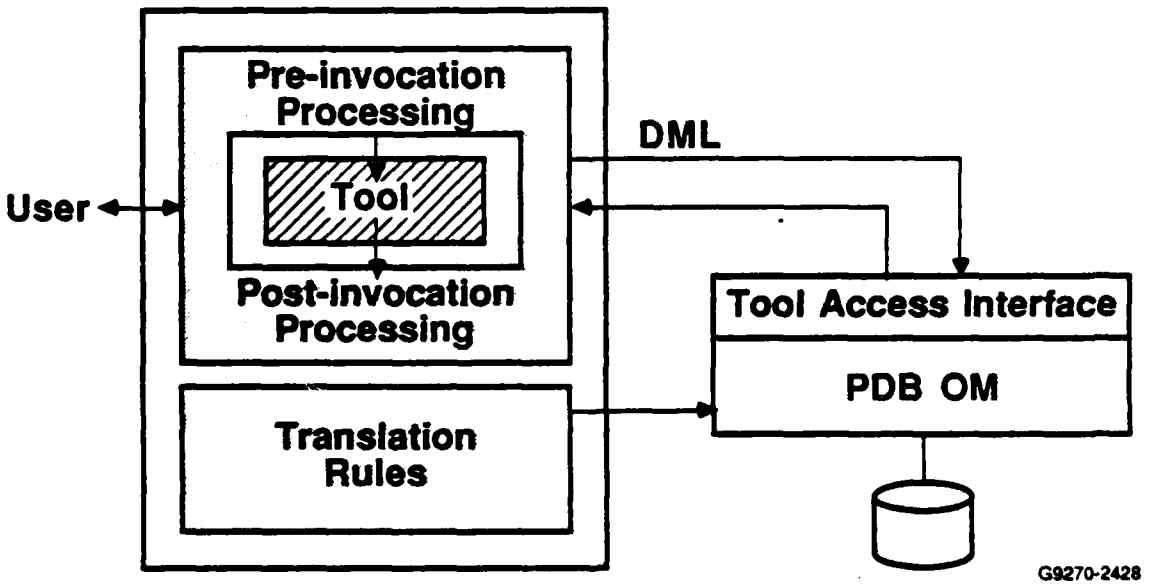


Figure 6. Indirectly-Coupled Tools

applications in the system to access data in the PDB object.

6.1.2. Tool/PDB Interfaces

The interfaces between tools and the PDB are shown in Figure 7. The Allocation and Reliability Analysis Tools, the CSCIs defined in Parts II and III of this document, are shown as directly-coupled tools. They will interface to the PDB directly through the Tool Access Interface, and associated with each tool will be a set of translation rules. Any UNIX or Cronus tools that become integrated will be indirectly-coupled tools, employing envelopes to interact with the PDB through the Tool Access Interface. Again, a set of translation rules will be developed for each envelope.

6.2. Database Schema Design Notes

This section documents some major PDB schema design decisions.

Editing

All objects may be "edited" by creating or modifying them with the PDB's native editor. Objects that have associated source code may also be edited with any traditional operating system supplied editor while objects that do not may be updated only with the PDB's updating mechanism. An object "has" source code when it has a method that generates an editable entity, that is source code, from its value and the values of some of its parts.

When an object is created or updated with a standard text editor, it must still be put back into the PDB as an object. This is accomplished with an interactive tool that asks the user how to break the text into slot fillers, and updates the PDB accordingly. It is not practical to develop a tool that puts objects back automatically because we allow objects with empty slots in the PDB. No practical software tool can determine which slot values are present and which are not without human help in the form of interactive responses to questions or structuring of the contents of the file by embedded directives.

Candidate Tools

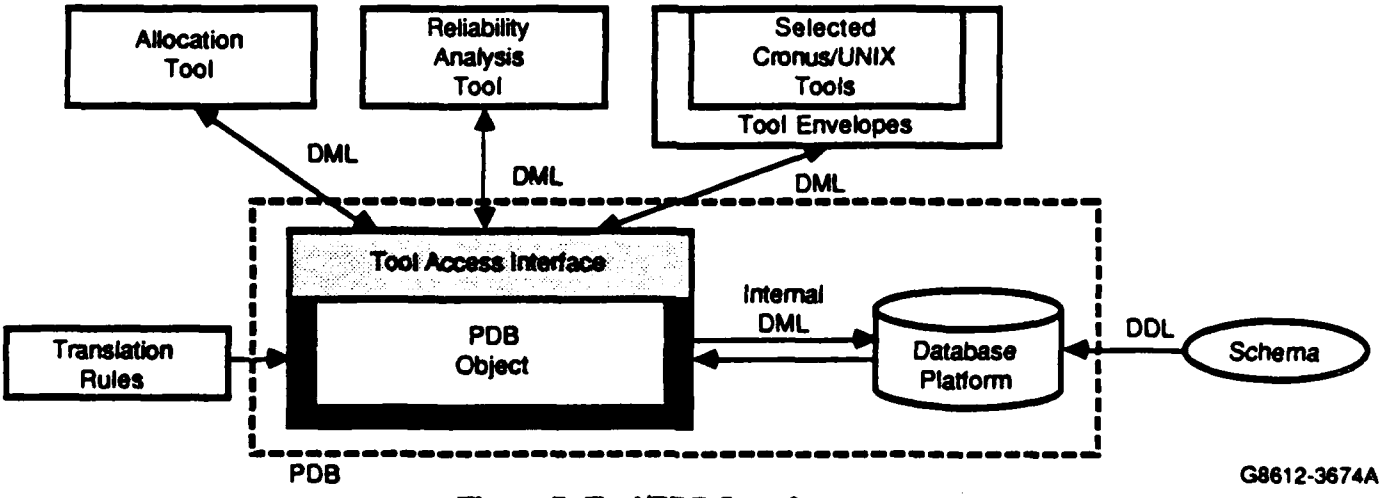


Figure 7. Tool/PDB Interfaces

The tools marked N are necessary to use the PDB; tools marked C make it more convenient.

1. (C) a type-library browser so developers can find things to reuse.
2. (N) an object entry tool to let people update objects that they have edited as text. Short term: An interactive tool that queries the user about what text to put in each slot. Long term: a tool that parses incomplete definitions and enters them.
3. (N) A translation engine and translation rules for every tool (existing and new) that is integrated into the PDB.

Replication of Cronus TDDB Information in PDB

Every type definition in the PDB replicates type information in Cronus's Type Definition Database (TDDB). This replication is necessary because software developers need information about a type's inheritance hierarchy in order to understand the type and that information is either not available from the TDDB or not accessible to tools and the PDB objects that need to use it. This redundancy does not seem to raise insurmountable problems with respect to the replicated entities.

6.3. Tool Integration Framework Requirements

This subparagraph provides a condensed version of the requirements specified for this CSCI. These requirements are derived directly from the System/Segment Specification document identified in paragraph 2.

The Tool Integration Framework CSCI will provide:

6.3.1. Project Database

The Project Database comprises the following components:

6.3.1.1. Information Model

The information model will:

- Contain the data types used by the supported tool set. (Supported tool set : Cronus tools TBD, a representative set of existing tools TBD, the Allocation tool and the Reliability Analysis tool).
- Be extensible (permit addition of new data types).

6.3.1.2. Data Model

The data model will:

- Be object oriented: support the definition of objects to represent the software/hardware entities in the system and ancillary documentation and management concepts such as work plan, properties that describe aspects of objects, and relations to other objects in the database to reflect the relationships between the various components of the software system under development. The model should support arbitrary directed and typed relations between objects.
- Allow the addition of new types (schema extension).
- Provide a data definition language (DDL) for defining types, specifying their placement in the type hierarchy, the operations they implement and inherit and the properties/attributes they possess.
- Allow the capture of dependencies and relationships among the data in the database through definition of relations among them.
- Impose a structure to facilitate inheritance. (This makes schema extension easier, as the common methods/properties/relations can be inherited by the new type from its parent type.)

6.3.1.3. Database Schema

- Database schemas will be defined and modified in a data definition language (DDL).
- A schema (or schemas) will be provided.

6.3.1.4. Data Definition Language (DDL)

(This DDL will be used to tailor (if necessary) a schema or create a new schema to conform to some project-specific requirements). Only static changes, i.e., changes to the schema BEFORE it is instantiated will be allowed. Dynamic modifications - modifications at run time AFTER the schema is instantiated, will not be allowed. Note: changes to objects will result in NEW object types. These new types will belong in the hierarchy as subtypes of the parent object type. The DDL will provide mechanisms:

- To create schemas.
- To define data types.
- To specify methods/interfaces to these data types.
- To specify operations/properties for data types.
- To create, delete or modify relations (explicitly) between objects

before a PDB instance is created.

6.3.1.5. Data Manipulation Language(DML)

The DML will provide mechanisms:

- To access data in the database.
- To traverse the instantiated database (if needed).
- To read and write attribute values associated with an object.
- To enforce data consistency and data integrity explicitly or implicitly.

6.3.1.6. User Interface

The user interface will:

- Permit the creation and instantiation of schemas.
- Provide an ad-hoc query facility to ask questions like "Who is the owner of this object?", "Which objects were created today?", "Which objects is this object *related* to?".
- Provide an on-line interactive context-sensitive (multi-level) help/advice capability that describes the syntax and function of commands/queries upon explicit request.

6.3.1.7. Cronus Interface

- Most of the Cronus manager development tools are integrated by the TDDB. To integrate these tools and make this information available to other tools, the data types in the TDDB or the entire TDDB will be contained in the PDB.
- Cronus *internals* will not be modified.

6.3.1.8. Database Services

The Database Services comprises of the Runtime Services and Schema Creation/Instantiation Services components.

6.3.1.8.1. Runtime Services

The Runtime Services component will:

- Traverse the instantiated database (in response to ad-hoc user queries and DML requests).
- Do requested operations on objects.
- Maintain relations among the objects.
- Create object instances in the instantiated database.

6.3.1.8.2. Schema Creation/Instantiation Services

This component will:

- Interpret DDL to create types and schemas.
- Instantiate schemas.

6.3.2. Database access support for foreign tools

Directly coupled tools use DML to access data directly. An example envelope will be provided for an indirectly coupled tool. This *envelope* will:

- Extract the required data items from the database;
- Transform the data form (if necessary) to be compatible with the tool's input needs;
- Invoke the tool;
- Transform the tool's output data (if necessary) to be compatible with the PDB;
- Update (write to the appropriate entities, create relations, specify properties) the PDB with the results of the tool's execution.

APPENDIX

Formal Schema Definition

This appendix provides a listing of the types in the formal schema for the Tool Integration Framework.

10.1. define type Environment;

supertypes = {Type};

properties = {

    name: String;
    Hardware-elements: distributed Set[Hardware];
    os-type: Operating-system;
    environment-for: Program-family;
    };

operations = /*none*/

    /* We do not have to define an iterator on Hardware-environment because the COP program statement

    iterate (<program-variable> = Hardware.class)

will yield elements of type Hardware, one at a time and we can test whether the element yielded is part of the Hardware-environment of interest. */

end Environment;

10.2. define type Operating-system;

supertypes = {Type}

properties = {

    name: String;
    }

operations = /*none*/

end Operating-system;

10.3. define type Hardware;

supertypes = {Type};

properties = {

    reliability: integer;
    MTBF: integer;
    };

operations = /*none*/

end Hardware;

10.4. define type Processor;

supertypes = {Hardware};

properties = {

name: String
relative-speed: Relative_Speed;
storage: Rational;
min: minload;
max: maxload;
mentioned-in: distributed Set[Object-constraints];
};

operations = /*none*/

end Processor;

10.5. define type Communication-subsystem;

supertypes = {Hardware};

properties = {

protocol: String; /*some computer program */
component: distributed Set[Communication-link]
};

operations = /*none*/

end Communication-subsystem;

10.6. define type Communication-link;

supertypes = {Type};

properties = {

to: Processor;
from: Processor;
cost: rational;
};

operations = /*none*/

end Communication-link;

10.7. define type Communication-delay-function;

supertypes = {Type};

properties = {

subsystem: Communication-Subsystem;
related-allocation: Allocation;
Delay: Rational;
};

operations = /*none*/

end Communication-delay-function;

10.8. define type Minload;

supertypes = {Type};

properties = {

value:rational;

};

operations = /*none*/

end Minload;

10.9. define type Maxload;

supertypes = {Type};

properties = {

value: rational;

}

operations = /*none*/

end Maxload;

10.10. define type allocation-constraint;

Design notes: This is the collection of arguments to a predication about allocating units of distribution to processors.

supertypes = {Type};

properties = {

mentioned-units-of-distribution: distributed Set[Unit-of-distribution];

occurs-in: distributed set[Allocation];

};

operations = /*none*/

end Allocation-constraint;

10.11. define type Object-constraint;

supertypes = {Allocation-constraint};

properties = {

mentions: distributed Set [Processor];

};

operations = /*none*/

end Object-constraint;

10.12. define type Load-constraint;

supertypes = {Allocation-constraint};

properties = {

min: Minload;

max: Maxload;

};

```
operations = /*none*/
end Load-constraint;
```

10.13. define type Program;

```
/* program is the system model*/
supertypes = {Type};
properties = {
      name: String;
      model: Program-model;
      requirements: String;
        /*relation of program to input-data shows up in Input-data-set type*/
        /*allocation shows up in the allocation object*/
        /*consists-of info shows up in UOD-as-used-...*/
      };
operations = /*none*/
end Program;
```

10.14. define type Input-data-set;

```
supertypes = {Type};
properties = {
      data: Text;
      owner-program: Program;
      };
operations = /*none*/
end Input-data-set;
```

10.15. define type Unit-of-Distribution;

```
supertypes = {Type};
properties = {
      name: string;
      i-output: Instrumentation-output;
      reliability: Rational;
      mentioned-in: distributed Set [Allocation-constraint];
      compiled-as-part-of: Program;
      };
operations = /*none*/
end Unit-of-Distribution;
```

10.16. define type allocated-U_O_D;

```
supertypes = {Type}
properties = {
      allocated-from: program;
      allocation-of: unit-of-distribution;
```

```
    comm-cost: communication-cost;
    }
operations = {
    }
end allocated-U_O_D;
```

10.17. define type Client;

```
supertypes = {Unit-of-Distribution};
properties = {
    skeleton: Text; /*the source-code that doesn't come from any of its parts*/
    source: Uninstrumented-Source-Code;
    isource: Instrumented-Source-Code;
    uses: distributed Set [mgr-file];
    };
operations = {
    collect-source (c:Client, Instrumented/Uninstrumented:Boolean)
     returns (Source)
     method ClientCollectSource
    };
end Client;
```

10.18. define type Type-manager;

```
supertypes = {Unit-of-Distribution};
properties = {
    spec: Type-manager-spec;
    };
operations = /*none*/
end Type-manager;
```

10.19. define type Type-manager-spec; supertypes = {Type};

```
properties = {
    name: String;
    abbrev: String;
    source: Uninstrumented-Source-Code;
    isource: Instrumented-Source-Code;
    };
operations = {
    collect-source (t:Type-manager-spec, Instrumented/Uninstrumented:Boolean)
     returns (Source)
     method TypeManagerSpecCollectSource
    };
end Type-manager-spec;
```

10.20. define type Mgr-file;

Design notes: This type holds each of genmgr's output files. We distinguish individual files for a particular type manager by their name attributes.

```
supertypes = {Type};
properties = {
        source: Uninstrumented-Source-Code;
        isource: Instrumented-Source-Code;
        name: String;
        manager: Type-manager;
        used-by: distributed Set [Client];
        };
operations = {
        collect-source (m:Mgr-file, Instrumented/Uninstrumented:Boolean)
         returns (Source)
         method MgrFileCollectSource
        };
end Mgr-file;
```

10.21. define type Program-module;

```
supertypes = {Type};
properties = {
        name: String;
        design-annotation: Text;
        source: Uninstrumented-Source-Code;
        isource: Instrumented-Source-Code;
        };
operations = {
        collect-source (p:Program-module, Instrumented/Uninstrumented:Boolean)
         returns (Source)
         method ProgramModuleCollectSource
        };
end Program-module;
```

10.22. define type Procedure;

```
supertypes = {Program-module};
properties = {
        source: Uninstrumented-Source-Code;
        isource: Instrumented-Source-Code;
        used-in: Client;
        };
operations = {
        collect-source (p:Procedure, Instrumented/Uninstrumented:Boolean)
         returns (Source)
         method ProcedureCollectSource
```

```
    };
end Procedure;
```

10.23. define type Program-model-element;

```
supertypes = {Type};
properties = {
    module: Program-module;
    model: Program-model;
    }
operations = /*none*/
end Program-model-element;
```

10.24. define type Program-model;

```
supertypes = {Type};
properties = {
    family: Program-family;
    program: Program;
    };
operations = /*none*/
end Program-model;
```

10.25. define type Program-family;

```
supertypes = {Type};
properties = {
    name: String;
    };
operations = /*none*/
end Program-family;
```

10.26. define type Source-code;

```
supertypes = {Type};
properties = {
    code: Text;
    };
operations = /*none*/
end Source-code;
```

10.27. define type Uninstrumented-Source-code;

Design notes: This type is here to model the fact that there are two disjoint types of source code, instrumented and uninstrumented. This has no properties other than those inherited from Source-code.

```
supertypes = {Source-code};
```

```
properties = /*none*/
operations = /*none*/
end Uninstrumented-Source-code;
```

10.28. define type Instrumented-source-code;

```
supertypes = {Source-code};
properties = {
    instrumentation-options: Text; /* tells what's instrumented*/
    };
operations = /*none*/
end Instrumented-source-code;
```

10.29. define type Executable-code;

```
supertypes = {Type}
properties = {
    code: Bit-string;
    name: String;
    executable-of: Unit-of-Distribution;
    composed-of: distributed Set [Object-code];
    }
operations = /*none*/
end Executable-code;
```

10.30. define type Object-code;

```
supertypes = {Type};
properties = {
    code: Bit-string;
    source: Source-code;
    options: distributed Set[Compiler-option];
    };
operations = /*none*/
end Object-code;
```

10.31. define type Compiler-option;

```
supertypes = {Type};
properties = {
    option: String;
    };
operations = /*none*/
end Compiler-option;
```

10.32. define type Type-spec;

Design notes: This is the form of the PDB's Type-def that Cronus genmgr uses.

```
supertypes = {Type};
properties = {
        name: String;
        constraints: Text;  /*this will be refined later; see genmgr*/
        source: Uninstrumented-Source-Code;
        isource: Instrumented-Source-Code;
        used-in: Type-manager-spec;
        derived-from: Type-def;
        };
operations = {
        collect-source (t:Type-spec, Instrumented/Uninstrumented:Boolean)
         returns (Source)
         method TypeSpecCollectSource
        };
        /* We need an iterator for Type-spec.class.*/
end Type-spec;
```

10.33. define type Type-def;

```
supertypes = {Program-module};
properties = {
        data-structure: Cantype;
        spec: Type-def-spec;
        number: Integer;
        abbrev: String;
        primal: Boolean;
        key: Text;  /*will be refined during detailed design*/
        replication: Text; /*will be refined during detailed design*/
        super: Type-def; /*for getting inheritance info*/
        };
operations =
        collect-source (t:Type-def, Instrumented/Uninstrumented:Boolean)
         returns (Source)
         method TypeDefCollectSource
/* a browsing tool will need get-Cronus-type-super and get-Cronus-type-sub */ };
end Type-def;
```

10.34. define type Type-def-operation;

```
supertypes = {Type};
properties = {
        name: String;
        body: Body;
        spec: Operation-spec;
```

```
    };
operations = {
    collect-source (t:Type-def-operation, Instrumented/Uninstrumented:Boolean)
     returns (Source)
     method TypeDefOperationCollectSource
    };
    /* We need an iterator for Type-def-operation.class */
end Type-def-operation;
```

10.35. define type Body;

```
supertypes = {Type}
properties = {
    source: Uninstrumented;
    isource: Instrumented;
    operation-cost: Invocation-Cost;
    }
operations = {
    }
end Body;
```

10.36. define type Type-def-spec;

```
supertypes = {Type};
properties = {
    data-invariant: Text;
    behavior-spec: Text;
    spec: distributed Set[Operation-spec];
    };
operations = /*none*/
end Type-def-spec;
```

10.37. define type Operation-spec;

```
supertypes = {Type};
properties = {
    head: header;
    pre: Text;
    post: Text;
    source: Uninstrumented-Source-Code;
    isource: Instrumented-Source-Code;
    returns: distributed Set[Parameter];
    };
operations = {
    collect-source (o:Operation-spec, Instrumented/Uninstrumented:Boolean)
     returns (Source)
```

```
        method OperationSpecCollectSource
      };
end Operation-spec;
```

10.38. define type Header;

```
supertypes = {Type};
properties = {
      name: Source;
      length: Integer;
      parameters: distributed Set[parameter];
      };
operations = /*none*/
end header;
```

10.39. define type Module Logical Call;

```
supertypes = {Type};
properties = {
      caller: Program-Module;
      callee: Program-Module;
      probability: real;
      criticality: integer;
      reliability: integer;

operations = /*none*/
end Call;
```

10.40. define type Operation Logical Call;

```
supertypes = {Type}
properties = {
      number-of-calls: Executive-Call-Number;
      callee: Type-Def-Operation;
      caller: Program-Module;
      logical-call: Module-Logical-call;
      }
operations = {
      }
end Operation-Logical-Call;
```

10.41. define type Execution-Call-Number;

```
supertypes = {Type}
properties = {
      value: integer;
      }
```

operations = {

}

end Execution-Logical-call

10.42. define type cost-component;

supertypes = {Type}

properties = {

invocation: Invocation-Cost;
call-number: Execution-Call-Number;
Comm-cost: communication-cost;
local-comm-cost: Local-Comm-Proc-Cost;
cost: real;
}

operations = {

}

end cost-component;

10.43. define type Invocation-cost;

supertypes = {Type}

properties = {

value: real;
}

operations = {

}

end invocation-cost;

10.44. define type Local-Comm-Proc-Cost;

supertypes = {Type}

properties = {

Cost: real;
comm-link-of: comm-Link;
header: header;
}

operations = {

}

end local-comm-proc-cost

10.45. define type Parameter;

supertypes = {Type};

properties = {

string: text;
occurs-in: distributed Set[Header];

used-in: distributed Set[Operation-spec];
};

operations = /*none*/

end Parameter;

10.46. define type Allocation;

supertypes = {Type};

properties = {

version: String;
has-performance: Performance;
/* assignments: distributed Set[Assignment] modeled in Assignment*/
constraints: distributed Set[allocation-constraint];
message-traffic: Integer;
delay: comm-delay-function;
associated-program: Program;
};

operations = /*none*/

end Allocation;

10.47. define type Performance;

supertypes = {Type}

properties = {

Reliability: Rational;
Response: Response-time;
Total: Total-time;
}

operations = /*none*/

end Performance;

10.48. define type Response-Time;

supertypes = {Type}

properties = {

value: rational;
max: processor-cost;
}

operations = {

}

end response-time

10.49. define type total-time;

supertypes = {Type}

properties = {

reliability: rational;
sum: distributed_set[Processor-cost];
}
operations = {
}
end total-time;

10.50. define type Assignment;

supertypes = {Type};
properties = {
platform: Processor;
software: allocated-U_O_D;
occurs-in: Allocation;
};
operations = /*none*/
/* We need an iterate operation for Assignment.class that will iterate through all the instances. */
end Assignment;

10.51. define type Error;

supertypes = {Type};
properties = {
name: String;
message: String;
used-by: Type-def;
};
operations = /*none*/
end Error;

10.52. define type Access-right;

supertypes = {Type};
properties = {
permission: String;
type: Type-def;
operation: Operation-spec;
};
operations = /*none*/
end Access-right;

10.53. define type Cantype;

supertypes = {Type};
properties = {

```
    name: String;
    rep: distributed Set[Representation];
    };
operations = /*none*/
end Cantype;
```

10.54. define type Representation;

```
supertypes = {Type};
properties = {
    rep-info: Text;
    };
operations = /*none*/
end Representation;
```

10.55. define type comm-delay-function;

```
supertypes = {Type}
properties = {
    delay: rational;
    }
operations = {
    }
end com-delay-function;
```

10.56. define type processor-cost;

```
supertypes = {Type}
properties = {
    }
operations = {
    }
end processor-cost
```

10.57. define type actual-residents-cost;

```
supertypes = {processor-cost}
properties = {
    reliability: rational;
    comm-cost: communication-cost;
    delay: network-delay;
    execution-cost:execution-cost
    }
operations = {
    }
```

end actual-residents-cost;

10.58. define type estimated-residents-cost;

supertypes = {processor-cost}

properties = {

cost: rational;
speed: relative-speed;
lifetime: lifetime-cost;
}

operations = {

}

end estimated-residents-cost;

10.59. define type execution-cost;

supertypes = {Type}

properties = {

}

operations = {

}

end execution cost;

10.60. define type type-def-exec-cost;

supertypes = {execution-cost}

properties = {

value:rational;
cost-component: distributed_set_of[execution-Cost-component];
}

operations = {

}

end type-def-execution-cost

10.61. define type client-execution-cost;

supertypes = {execution-cost}

properties = {

value: rational;
}

operations = {

}

end client-execution-cost;

10.62. define type execution-cost-component;

```
supertypes = {Type}
properties = {
        invocation-cost: invocation-cost;
        number-of-calls: execution-call-number;
        }
operations = {
        }
end execution-cost-component;
```

10.63. define type lifetime-cost;

```
supertypes = {Type}
properties = {
        cost: rational;
        }
operations = {
        }
end lifetime-cost;
```

10.64. define type network-delay;

```
supertypes = {Type}
properties = {
        delay: rational;
        }
operations = {
        }
end network-delay;
```

10.65. define type communication-cost;

```
supertypes = {Type}
properties = {
        value: rational;
        }
operations = {
        }
end communication-cost;
```

PART II

ALLOCATION TOOL

1. Scope

1.1. Identification

This Software Top Level Design Document describes the top level design for the Computer Software Configuration Item (CSCI) identified as the Allocation Tool, CSCI Number 2, of the Tool Integration Framework and Development Tools System. The requirements for the System appear in the Draft System/Segment Specification.

1.2. Overview

This overview presents a high-level view of the Allocation Tool, including purpose, role in the integrating framework, basic operation and development plan.

1.2.1. Purpose

Distributed application developers must, at some point during development, address the question of how to assign the program's units of distribution (objects and clients) to the processing nodes. While the manner in which they are assigned will not affect the application's functionality, it will have major implications for objectives related to performance, fault tolerance, resource utilization and security. Assignments will vary with respect to how the program performs during execution since resource utilization profiles, communication loads, and parallelism are all influenced by the pattern of allocation. The allocation problem is an important concern in DISE where performance and fault-tolerance are critical objectives for distributed applications.

Unfortunately for developers, the allocation problem exhibits combinatorial complexity; finding the optimal assignment of components to nodes is an inherently difficult problem. Subtle tradeoffs exist between resource utilization, parallelism and communication costs. Nevertheless it remains an important issue in the development of distributed applications in this environment.

The Allocation Tool supports distributed application developers by determining an efficient assignment of program units to processing nodes. The tool utilizes information about the distributed application, its components and their interaction patterns, as well as characterizations of the processing environment. It balances processing and communication costs in finding an appropriate processor assignment for each unit of distribution in the program.

1.2.2. Role in the Integration Framework

Besides assisting application developers with determining an efficient program allocation, there is a second major purpose behind the Allocation Tool CSCI. The Allocation Tool serves as an illustration of the Tool Integration Framework, the CSCI defined in Part I of this document. As an example of a tool that supports distributed application developers, the Allocation Tool will be integrated, as a directly-coupled tool, into DISE through the Tool Integration Framework CSCI.

The Allocation Tool accesses the PDB for its input and output data requirements. As shown in Figure 8, the tool is implemented as a Cronus client program offering a window-based user interface. This user interface features a tool-specific interface to the PDB which permits the user to control tool data I/O via the PDB. The tool uses the DOS's interprocess communication facilities to interact with the PDB over the network, thereby allowing the tool and application developer to have transparent access to the PDB over the network.

1.2.3. Use and Functionality

The Allocation Tool can be applied across the stages of development, from early in development to implementation. The basic information the user must supply includes attributes of the processing

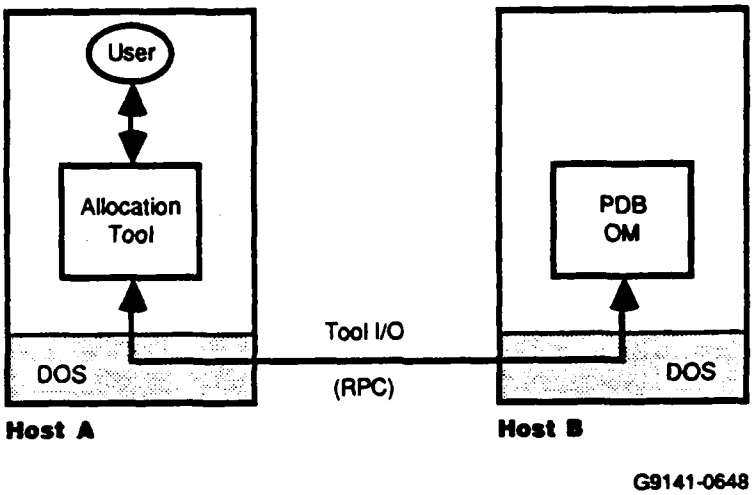


Figure 8. Allocation Tool and DISE

environment and information about the distributed application. Typically the processing environment is a known quantity and information about it can be fetched from the PDB, while the application represents the variable parameter.

Early in development the tool can be used to evaluate alternative high-level program decompositions with respect to their resource utilization. At this stage, the developer may only have some initial decompositions into objects and clients along with some rough estimates of their resource utilization. The user provides this information to the tool, either directly or via the PDB. The tool then computes and displays for the user an allocation and its associated performance.

During implementation the tool is used to optimize the program's runtime performance. More is known about the program at this stage, and the tool accommodates the additional detail. Again the user may provide the data directly or may just have the tool fetch it from the PDB. The tool computes and displays an allocation based on the specified inputs.

The tool allows the user to manipulate any of the input parameters in order to investigate its effect on program performance. For example, the user may add some additional objects to the application, or may change the resource utilizations of an object or the nature of interactions between two objects. Experimentation is an important capability to offer developers. This tool supports experimentation by offering generality and flexibility with respect to the types of inputs accepted and by providing a convenient interface through which they can be manipulated.

A simple loop captures the main functionality of the tool: collect all the input data, find an efficient allocation, and display the output. Finding an efficient allocation involves two tasks: cost model construction and optimization. The former task entails constructing a cost model that reflects the expected performance of the distributed application as a function of a particular allocation scheme. Factors incorporated into the cost model include attributes of both the application and the processing environment. The cost model captures execution costs of application modules, intermodule communication costs and parallelism between modules. The optimization component employs a search heuristic to find an efficient allocation amongst all possible allocations.

1.2.4. Development Phases

Development of the Allocation Tool CSCI will take place in four phases:

Phase 1:

Develop a convenient user interface. The user interface will offer window-based data-entry facilities. In addition, it will provide a tool-specific interface to the Project Database for retrieval and storage of tool data at the user's discretion. Completing this phase will allow the tool to be integrated with the PDB when the PDB becomes operational, even though the tool will exhibit minimal functionality.

Phase 2:

Implement basic tool functionality. This will include a simple form of the cost model component that does not consider communication costs. A simple search heuristic will be employed in the optimization component.

Phase 3:

Extensions to the cost model to include communication costs in the performance computation, and enhancements to make the search algorithm more effective.

Phase 4:

Refinements to the cost model to make it more accurate and complete and to the search algorithm component to increase its efficiency.

Phase 3 represents the target implementation for this tool. The final phase is optional, and will be entered only if sufficient time and effort are available for it.

1.3. Introduction

This document presents a top level functional and abstract data type decomposition of the Allocation Tool CSCI into Top-level Computer Software Components (TLCSCs) and the interfaces among TLCSCs of this CSCI. It also defines the interfaces between this CSCI and other elements of the Tool Integration Framework and Development Tools System. Included in the Notes (section 6) are an external interface definition and a detailed requirements specification derived from the System/Segment Specification.

2. Referenced Documents

- *Draft System/Segment Specification*, DOS Design Application Tools, Contract No. F30602-87-C-0104, CDRL No. 3, 25 August, 1988, RADC/COTD, (Honeywell Inc., Corporate System Development Division).

3. Requirements

3.1. CSCI Architecture

Figure 9 shows the top level functional decomposition for the Allocation Tool CSCI. The elements of the decomposition, each a TLCSC, are functions and abstract data types (ADTs). The primary control loop is captured in the function Main. The Cost_Model ADT encapsulates the application cost model, the Allocation ADT encapsulates the optimization algorithm and the UI ADT encapsulates the interfaces to the user and the PDB.

3.2. Functional Allocation

The requirements for this CSCI are defined in the System/Segment Specification. In particular, the requirements for the Allocation Tool specified in section 3.1.6.2 of the System/Segment Specification

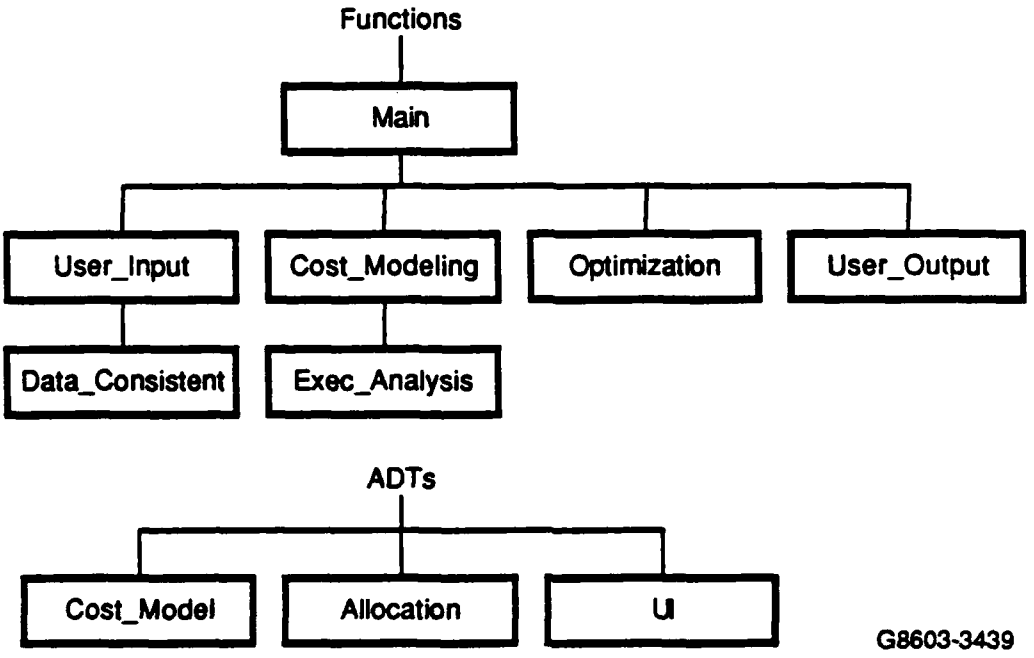


Figure 9. Allocation Tool Functional Decomposition

identify requirements for three functions: cost model generator, optimizer, and user interface. Identified below are the TLCSCs to which each of the three functions has been allocated:

- Cost model generator: This is implemented by function TLCSCs 3.6.1.3 (Exec_Analysis), 3.6.1.4 (Cost_Modeling), and ADT TLCSC 3.6.2.1 (Cost_Model).
- Optimizer: This is implemented by function TLCSCs 3.6.1.5 (Optimization) and ADT TLCSCs 3.6.2.1 (Cost_Model) and 3.6.2.2 (Allocation).
- User_Interface: This is implemented by function TLCSCs 3.6.1.1 (User_Input), 3.6.1.2 (Data_consistent), 3.6.1.6 (User_Output) and ADT TLCSC 3.6.2.3 (User_Interface).

3.3. Memory and Processing Time Allocation

Not specified.

3.4. Functional Data Flow

The functional data flow for the Allocation Tool is shown in figure 10. Boxes in the diagram represent individual function and ADT TLCSC's, the latter occupying the bold-lined boxes. Arrows between boxes indicate data flow, and have been annotated to show the primary data that is passed from one TLCSC to another.

Note that the UI (user interface) ADT encapsulates both the user and the PDB. The tool user therefore interacts with precisely one TLCSC, which also serves as the interface to the PDB. Data flow from

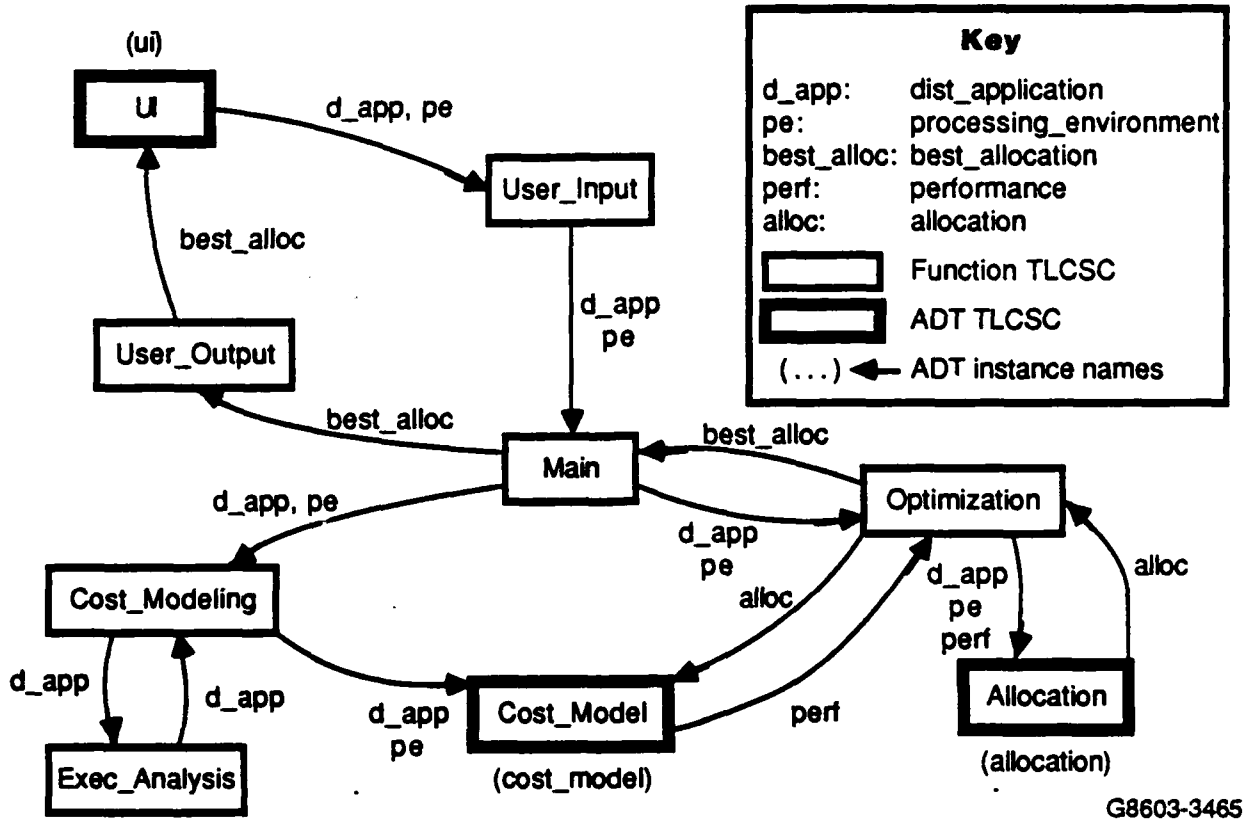


Figure 10. Allocation Tool Data Flow

the UI ADT, therefore, may ultimately be coming from either the user or the PDB.

3.5. Global Data

3.5.1. Global Data Types

```
App_Type = (Names, Specs, Code);
Cost_Type = (Obj_Wkld, Opn_Wkld, Input_Data);
Comp_Type = (Client, Object);

Opn_Headers = RECORD
    name : STRING;
    params : LIST OF Types;
    return_type : Type;
END;
```

```
Opn_List = SET OF Opn_Headers;

Cost_Rec = RECORD (* costs associated with an application component *)
    obj_wkld : Exec_Time;  (* total workload *)
    opn_wkld : ARRAY [Opn_Names] OF Exec_Time; (* workload per invocation *)
    call_profile : ARRAY [Appl_Comps] OF
        ARRAY [Opn_List] OF CARDINAL; (* frequency *)
END;

Components = RECORD     (* information about a client or object *)
    comp_name : STRING; (* Key for accessing components. *)
    comp_kind : Comp_Type;
    opns : Opn_List; (* Specs and Code App_Types *)
    ext_spec_ref,
    ext_body_ref: External_File; (* Code App_Type only *)
    cost_data : Cost_Rec;
END;

Appl_Comps = SET OF Components;

Dist_Appl_Type = RECORD (* a distributed application *)
    kind : App_Type;
    comps : Appl_Comps;
    cost : Cost_Type;
    data_set : Input_Data_Spec;
END;

Processor_Type = RECORD
    name : STRING;
    relative_speed : Fraction;
END;

Proc_Hosts = SET OF Processor_Type;

Proc_Env_Type = RECORD
    processors : Proc_Hosts;
    comm_cost  : MATRIX [Processor_Type, Processor_Type] OF
        Comm_Cost_Func(msg_size : CARDINAL); (*F in the Cost_Model ADT*)
    comm_delays : Comm_Delay_Func( allocation : Alloc_Type;
            msg_traffic : CARDINAL ); (*D in the Cost_Model ADT*)
    loading_factor : Network_Congestion_Func(
            load : SET OF Components;     (* local objects/clients *)
            threads_of_control : CARDINAL (* number of clients *)
            );  (*g in the Cost_Model ADT*)
END;

Obj_Func = (Response_Time, Total_Time);

Allocation_Constraints = ARRAY [1..1000] OF RECORD
```

```
    num_constraints : 0..1000;
    constraint_type = (Object_Constraint, Load_Constraint);
    CASE constraint_type OF
        Object_Constraint :
            object_name : STRING;
            targets : Proc_Hosts;
        Load_Constraint :
            proc_name : STRING;
            max_load,
            min_load : CARDINAL;
    END;
END;

Search_Constraint = RECORD
    cpu_time_limit,
    real_time_limit : System_Time;
    max_evaluations : CARDINAL;
    threshold : Execution_Time;
END;

MAPPING FROM SET X TO SET Y = (* Total function from X to Y.
    Abstract model: a set of pairs <x,y>.
    Operations: For x in X and y in Y:
        assign(x, y) : associates y with x.
        lookup(x) returns y : returns associated y for x. *)

An_Allocation = MAPPING FROM Appl_Comps TO Proc_Hosts;

Allocation_Info = RECORD
    assign : An_Allocation;
    perf : Exec_Time;
END;

Command = (def_app, def_costs, def_proc_env, def_objfunc, def_ass_constr,
        def_search_constr, continue);
```

3.5.2. Global Variables

```
(*** INPUT VARIABLES ***)
dist_application : Dist_Appl_Type; (* The distributed application under study. *)
processing_environment : Proc_Env_Type; (* The hardware environment *)
objective_function : Obj_Func; (* The choice of objective function. *)
assign_constraints : Allocation_Constraints; (* The assignment constraints. *)
search_constr : Search_Constraint; (* The 4 different search algorithm constraints. *)

(*** ADT INSTANCES ***)
cost_model : Cost_Model; (* The cost model ADT instance. *)
```

allocation : Allocation; (* The allocation search ADT instance. *)

ui : UI; (* The user interface. *)

(*** GLOBAL DATA STRUCTURES ***)

best : Allocation_Info; (* The current best allocation found. *)

finished : BOOLEAN; (* Termination flag. *)

3.6. Top Level Design

3.6.1. Function TLCSCs

3.6.1.1. Main TLCSC

3.6.1.1.1. Identification

This is the main function in the CSCI. It encapsulates the primary control loop for the tool. It repeats the following four tasks until the user terminates the program: get input data, construct a cost model, find an allocation and display it to the user.

3.6.1.1.2. Inputs

There are no inputs to this function. In the case of this function, inputs would be command-line arguments provided at the time of tool invocation.

3.6.1.1.3. Local Data

Local data for the main program is global data for the tool, and global data is identified in section 3.5.2 above.

3.6.1.1.4. Processing

```
BEGIN Main;

    LOOP
        User_Input(dist_application, objective_function, processing_environment,
            assign_constraints, search_constr, ui);
        Cost_Modeling(dist_application, objective_function, processing_environment, cost_model);
        Optimization(dist_application, processing_environment, assign_constraints,
            search_constr, cost_model, best);
        User_Output(dist_application, processing_environment, search_constr, best, ui, finished);
    UNTIL finished;

END Main.
```

3.6.1.1.5. Outputs

No outputs are generated by this function within the CSCI. The external interface generated, in essence, by this function is described in section 6.1.

3.6.1.2. User_Input TLCSC

3.6.1.2.1. Identification

This function uses a user-interface ADT to obtain input parameters from the user. The user can provide input items in any order; the user explicitly indicates that they are finished supplying inputs. At that time, consistency checks are made to ensure that data provided is complete and consistent.

3.6.1.2.2. Inputs

This function takes no inputs.

3.6.1.2.3. Local Data

```
done : BOOLEAN; (* end-of-input flag *)
```

3.6.1.2.4. Processing

```
BEGIN User_Input;

    ui := UI$init();                    (* Initialize *)
    objective_function := Response_Time;
    assign_constraints.num_constraints := 0;
    search_constr.* := INFINITY;
    done := FALSE;
    LOOP                                (* Process a command *)
        CASE UI$command(ui) OF
            def_app: (* update dist_application info *)
                <dist_application.kind, dist_application.comps> :=
                    UI$get(ui, <App_Type, Appl_Comps>);
            def_costs: (* update cost_data *)
                <dist_application.cost, dist_application.data_set> :=
                    UI$get(ui, <Cost_Type, Input_Data_Spec>);
            def_proc_env: (* update processing environment *)
                processing_environment := UI$get(ui, Proc_Env_Type);
            def_objfunc: (* objective function = Resp_Time or Exec_Time *)
                objective_function := UI$get(ui, Obj_Func);
            def_ass_constr: (* add/deleted from list of constraints *)
                assign_constraints := UI$get(ui, Allocation_Constraints);
            def_search_constr: (* establish any of 4 constraints *)
                search_constr := UI$get(ui, Search_Constraint);
            continue:
                IF Data_consistent(dist_application, processing_environment, assign_constraints)
                THEN
                    done := TRUE;
                ELSE
                    UI$error(ui, "Data inconsistency");
                END;
        END;
    UNTIL done;

END User_Input;
```

3.6.1.2.5. Outputs

dist_application : Dist_Appl_Type;
objective_function : Obj_Func;
processing_environment : Proc_Env_Type;
assign_constraints : Allocation_Constraints;
search_constr : Search_Constraint;
ui : UI;

3.6.1.3. Data_consistent TLCSC

3.6.1.3.1. Identification

This function checks the user-supplied tool input values for consistency and completeness.

3.6.1.3.2. Inputs

dist_application : Dist_Appl_Type;
processing_environment : Proc_Env_Type;
assign_constraints : Allocation_Constraints;

3.6.1.3.3. Local Data

None.

3.6.1.3.4. Processing

BEGIN Data_consistent;

(* Check for consistency between input data items and that sufficient inputs have been provided. Required inputs are the distributed application, application costs and processing environment. Returns TRUE if data is consistent and complete, FALSE otherwise. *)

END Data_consistent;

3.6.1.3.5. Outputs

RETURNS BOOLEAN;

3.6.1.4. Exec_Analysis TLCSC

3.6.1.4.1. Identification

This function compiles an instrumented version of the application and executes it on a pre-designated host processor. It captures cost data about the application, including operation execution times and call profiles, according to the Cost_Rec data type.

3.6.1.4.2. Inputs

dist_application : Dist_Appl_Type;

3.6.1.4.3. Local Data

None.

3.6.1.4.4. Processing

BEGIN Exec_Analysis;

(* Execute an instrumented version of the distributed application to obtain cost measures. Source code and input data are specified in **dist_application**. The empirical measures generated will be stored in **dist_application.cost**.

If instrumentation tools are available in the development environment, the implementation of this function will simply involve generating an appropriate workload profile that can be fed to the instrumentation tool and then invoking the tool to obtain cost measures. The PDB would serve as the mechanism by which the workload and cost measures are transmitted between the two tools. *)

END Exec_Analysis;

3.6.1.4.5. Outputs

dist_application : Dist_Appl_Type; (* with updated cost component *)

3.6.1.5. Cost_Modeling TLCSC

3.6.1.5.1. Identification

This function generates a cost_model, which is an instance of the Cost_Model ADT. The cost model is based on the application cost attributes, which may have to be generated by performing the execution analysis function, and processing environment attributes.

3.6.1.5.2. Inputs

dist_application : Dist_Appl_Type;
objective_function : Obj_Func;
processing_environment : Proc_Env_Type;

3.6.1.5.3. Local Data

None.

3.6.1.5.4. Processing

BEGIN Cost_Modeling;

IF dist_application.cost = Input_Data THEN
    Exec_Analysis(dist_application);
END;
cost_model := Cost_Model$generate (objective_function, dist_application, processing_environment);

END Cost_Modeling;

3.6.1.5.5. Outputs

dist_application : Dist_Appl_Type;
cost_model : Cost_Model;

3.6.1.6. Optimization TLCSC

3.6.1.6.1. Identification

This function uses Cost_Model ADT to find a "good" allocation of application components to processors. An Allocation ADT instance is generated which encapsulates the optimization algorithm. The function repeatedly gets a new allocation from the Allocation ADT and analyzes its performance using the Cost Model ADT until either the optimization algorithm signals termination or user-specified termination criteria are met. The best allocation found is returned.

3.6.1.6.2. Inputs

```
dist_application : Dist_Appl_Type;
processing_environment : Proc_Env_Type;
assign_constraints : Allocation_Constraints;
search_constr : Search_Constraint;
cost_model : Cost_Model;
```

3.6.1.6.3. Local Data

```
temp : Allocation_Info; (* Temporary variable. *)
count : CARDINAL; (* The number of allocation evaluations made. *)
done : BOOLEAN;
```

3.6.1.6.4. Processing

```
BEGIN Optimization;

    done := FALSE;
    allocation := Allocation$initialize(dist_application.comps, processing_environment.processors,
        assign_constraints, search_constr);
    best.assign := Allocation$mapping(allocation);
    best.perf := Cost_Model$evaluate(cost_model,best.assign);
    count := 1;
    allocation := Allocation$set_perf(allocation, best.perf);
    WHILE  (NOT done) AND (best.perf > search_constr.threshold) AND
        (cpu_time() < search_constr.cpu_time_limit) AND
        (clock_time() < search_constr.real_time_limit) AND
        (count < search_constr.max_evaluations)
    DO
        allocation := Allocation$generate(allocation); (* search algorithm iteration *)
        temp.assign := Allocation$mapping(allocation);
        temp.perf := Cost_Model$evaluate(cost_model,temp.assign);
        INC(count);
        allocation := Allocation$set_perf(allocation, temp.perf);
        IF temp.perf < best.perf THEN best := temp; END;
    END_WHILE;
    EXCEPTION HANDLER(CONVERGED);
        done := TRUE;
    END EXCEPTION HANDLER;

END Optimization;
```

3.6.1.6.5. Outputs

best : Allocation_Info;

3.6.1.7. User_Output TLCSC

3.6.1.7.1. Identification

The best allocation and its performance are presented to the user, who can then decide to either quit or continue with another iteration of the tool.

3.6.1.7.2. Inputs

dist_application : Dist_Appl_Type;
proc_env : Proc_Env_Type;
s_constr : Search_Constraint;
best : Allocation_Info;
ui : UI;

3.6.1.7.3. Local Data

None.

3.6.1.7.4. Processing

BEGIN User_Output;

ui := UI$display(ui, best, dist_application, proc_env, s_constr);
finished := UI$query(ui); (* "quit or continue?" *)

END User_Output;

3.6.1.7.5. Outputs

ui : UI;
finished : BOOLEAN;

3.6.2. ADT TLCSCs

3.6.2.1. Cost_Model ADT TLCSC

3.6.2.1.1. Identification

The cost_model ADT implements the allocation cost model according to the objective function requested. It is based on attributes of the application (components and their costs) and processing environment attributes. The **generate** operation creates a cost model instance, and the **evaluate** operation computes the cost of a given allocation.

3.6.2.1.2. Abstract Model

Cost_Function : ARRAY [An_Allocation] OF Exec_Time;

(* each allocation has a corresponding performance *)

3.6.2.1.3. Operations

generate: ob_fun: Obj_Func; app: Dist_App_Type; p_e: Proc_Env_Type --> Cost_Model;

(* This operation generates the application cost model used to evaluate allocation of dist_appl components to proc_env hosts. The cost model is expressed as a function of an allocation (of type An_Allocation). An allocation maps application components to processors, and can be expressed as a function $A : Appl\_Comps \rightarrow Proc\_Hosts$. We assume in the following definitions that subscript indices uniquely identify members of the set of application components, object operations and processors. These sets are denoted as follows:

Ω = set of application components (app.comps)

Θ_i = set of operations on object i (app.comps[i].opns)

Γ = set of processors (p_e.processors)

The term $P_n(a), a \in A$, expresses the processing cost for processor n as a function of a given allocation a. There are two forms for the objective function, each expressed in terms of P_n for a given allocation $a \in A$:

$$MAX\{ P_n(a) : 1 \le n \le |\Gamma| \} \qquad \text{(Response Time)}$$

$$\sum_{n=1}^{|\Gamma|} P_n(a) \qquad \text{(Total Time)}$$

For convenience in exposition, an assignment variable is introduced:

$$X_{in} = \begin{cases} 1 & a(i) = n \\ 0 & otherwise \end{cases}$$

P_n is defined as the sum of the execution and communication costs of the application clients and objects assigned to it:

$$P_n(a) = EXEC_n(a) + COMM_n(a) + NETDELAY_n(a)$$

$EXEC_n$ denotes resident application component (object and client) execution costs. $COMM_n$ denotes the communication costs incurred by those components, and $NETDELAY_n$ is the cost experienced by host n due to network delays. The form of these terms depends on the type of cost information provided by the user. The two cost types are referred to as Object workload and Operation Workload. For Object Workload cost information (app.cost = Obj_Wkld) we have:

$$EXEC_n(a) = \sum_{i=1}^{|\Omega|} [\, X_{in}\, E_i(n) \,]$$

$$COMM_n(a) = 0$$

$$NETDELAY_n(a) = 0$$

E_i denotes the expected computational workload for object i over its lifetime, and is a function of attributes of the host processor n. This term will come from app.comps[i].cost_data.obj_wkld and will be weighted according to the relative speed of processor n as given in p_e.processors[n].relative_speed. Communication costs under this cost model are 0, since no explicit information about the interactions between objects is available.

In the case of Operation Workload (app.cost = Opn_Wkld) $EXEC_n$, $COMM_n$ and $NETDELAY_n$ are expressed as:

$$EXEC_n(a) = \sum_{i=1}^{|\Omega|} \sum_{k=1}^{|\Theta_i|} \left[X_{in}\ E_{ik}(n) \sum_{j=1}^{|\Omega|} C_{jik} \right]$$

$$COMM_n(a) = \sum_{i=1}^{|\Omega|} \sum_{j=1}^{|\Omega|} \sum_{m=1}^{|\Gamma|} \sum_{k=1}^{|\Theta_j|} \left[X_{in}\ X_{jm}\ C_{ijk} \left[F(n,m,S_{jk}) + E_{jk}(m)\ (1 - X_{jn})\ g_T(a^{-1}(n)) \right] \right]$$

$$NETDELAY_n(a) = D_{C,S}(n, a)$$

E_{ik} denotes the computational workload for operation k of object i for an arbitrary invocation, and is a function of the host processor n's attributes. This quantity comes from app.comps[i].cost_data.opn_wkld[k] and will be weighted according to the relative speed of processor n as given in p_e.processors[n].relative_speed.

C_{ijk} denotes the number of calls object i makes on operation k of object j. It is obtained from app.comps[i].cost_data.call_profile[j][k]. S_{jk} denotes the size in bytes of the message contained in the call to operation k on object j. This is derived from app.comps[j].opns[k].params.

The term $a^{-1}(n)$ is the inverse image of the function a. It identifies the set of objects and clients assigned to node n:

$$a^{-1}(n) = \{\ i\ :\ a(i) = n\ \}$$

$F(n,m,S_{jk})$ is the local processing cost for node n issuing a call to an object on node m of size S_{jk} and corresponds to the p_e.comm_cost input parameter.

The factor $g_T(a^{-1}(n))$ where T indicates the number of threads of control in the distributed application, is in the range [0,1] and comes from p_e-loading_factor input parameter. It reflects the degree to which a processor experiences communication delays due to the remote execution of an RPC on behalf of a resident task. It is 1 if there is a single thread of control in the application (since only 1 application component can execute at any time). It tends towards 0 as the number of threads and the number of local objects increases.

$D_{C,S}(n, a)$, given by p_e.comm_delays, is the communication network delay. A given number of calls in the system (C) and the sizes of the messages for those calls (S) results in a cost to processor n under the application allocation a. *)

evaluate: Cost_Model, An_Allocation --> Exec_Time;

(* Returns the expected execution time under an allocation by evaluating it according to the cost function defined for Cost_Model. *)

3.6.2.2. Allocation ADT TLCSC

3.6.2.2.1. Identification

The Allocation ADT encapsulates the search algorithm used to find an optimal allocation. Following initialization using the **initialize** operation, successive iterations of the search algorithm are effected by calling the **generate_allocation** operation. Prior to generating an allocation the **set_perf** operation is invoked to provide the search algorithm with the performance of an allocation. The actual mapping of

application components to processors hidden within an Allocation instance is obtained using the **mapping** operation.

3.6.2.2.2. Abstract Model

current_allocation : Allocation_Info; (* mapping + performance *)
search_state : Search_State_Information;
performance : Real;
search_state : Search_State_Information;
assign_constraints : Allocation_Constraints;

3.6.2.2.3. Operations

initialize: Appl_Comps, Proc_Hosts, Allocation_Constraints, Search_Constraint --> Allocation;

(* The generator. Initializes the search algorithm. Search constraints may affect algorithm chosen. Allocation constraints are satisfied by the Allocation generated.

One quick and easy approach for this operation would be to employ a greedy technique. Application components are ranked in order of decreasing cost, ignoring communication, to processors that are ranked according to processing power. Allocation constraints are met when each component is assigned. *)

generate_allocation: Allocation --> Allocation;

(* Generates a new Allocation based on the state of the previous Allocation according to the search algorithm employed. That is, it generates the next iteration of the search algorithm.
RAISES CONVERGED: this indicates that the algorithm has reached convergence. This does NOT mean that the mapping generated is optimal, but simply that no more improvements are possible.
RAISES NO_PERFORMANCE: this indicates that the input Allocation has not been assigned a performance value. A performance value may be required by the search algorithm employed.

A hill-climbing technique may be employed here. On each call the previous allocation is modified according to some deterministic modification scheme. If, on any call, improvement is made, then the algorithm is repeated. When a complete round has been made with no improvement, the algorithm terminates, raising the CONVERGED condition. This approach is easy to implement, each call is quick and it is guaranteed to terminate. However, it may fail to reveal a good solution by getting stuck on a local optimum.

Another approach would use simulated annealing. It generates a new allocation based on the previous one. This may be somewhat more difficult to implement than hill-climbing, but is less susceptible to the local optimum problem. Each call is quick but the approach lacks inherent termination criteria. *)

set_perf: Allocation, Exec_Time --> Allocation;

(* Sets the value of the current allocation in Allocation to performance_value. *)

mapping: Allocation --> An_Allocation;

(* Returns the current allocation of Allocation. *)

3.6.2.3. UI ADT TLCSC

3.6.2.3.1. Identification

This ADT encapsulates the user interface and the interface to the PDB. It manages the display device and supports mouse and keyboard input devices. It handles the tool's I/O needs by getting data either from the user directly or from the PDB, as commanded by the user.

The user interface is created using the **init** operation, **command** gets a user command, **get** gets a data item either from a file, the PDB or directly from the user, **display** and **error** display information for the user, and **query** determines if the user is finished.

3.6.2.3.2. Abstract Model

screen : User_Interface_Display;

3.6.2.3.3. Operations

init: --> UI;

(* The generator. Initializes the user interface. *)

command: UI --> Command;

(* Returns a user command. *)

get: UI, Type --> Type;

(* Generic operation which returns a value for the specified type obtained from the user, perhaps through the PDB. *)

error: UI, STRING --> UI;

(* Returns a user command. *)

display: UI, Allocation_Info, Dist_Appl_Type, Proc_Env_Type, Search_Constraint
--> UI;

(* Displays the allocation from Allocation_Info, along with the application and processing environment, in an appropriate graphical manner. *)

query: UI --> BOOLEAN;

(* Returns TRUE if user is finished, FALSE otherwise. *)

3.7. Adaptation Data

This tool CSCI executes on a Sun workstation and uses the Sunview windowing system for the user interface. Since the tool is developed as a Cronus application, the host workstation environment must run the Cronus distributed operating system.

6. Notes

6.1. Interfaces

As an application development tool, the Allocation Tool interfaces with the following external entities:

- Cronus distributed operating system: The tool, like any Cronus application, may require access to various Cronus system object managers.
- Project Database: The PDB is a specific Cronus object manager to which the Allocation Tool interfaces for various I/O needs. The data items potentially obtained from the PDB include information about the components in the distributed application, target processing environment

attributes and assignment constraints which an allocation must meet. The best allocation and its performance as generated by the tool may be stored in the PDB. The interface itself is defined by the operations exported by the PDB object.

6.2. Requirements Specification

This subparagraph provides a condensed version of the requirements specified for this CSCI. These requirements are derived directly from the Draft System/Segment Specification identified in paragraph 2.

6.2.1. Purpose

The Allocation tool accepts input data through a convenient user interface and generates an output consisting of an allocation of application components to processing sites and that allocation's expected performance.

6.2.2. Inputs

6.2.2.1. Distributed application

A distributed application is defined as one or more client modules and the set of objects to which the clients pass messages, directly or indirectly. Objects consist of two parts, a specification and a body. Clients are programs which invoke objects. One of the following three forms of description of a distributed application will be provided:

1) Definitions of the objects and clients in the application. This will consist of just the names of all clients and objects, no code.

2) Client names and object specifications. Object specifications contain headings for procedures and functions (that is, declarations of the object's methods, including parameters and their types) but have no corresponding bodies.

3) Source code for the application. Objects will have both specifications and bodies; clients will be completely coded.

6.2.2.2. Distributed application cost estimates

One of the following three forms of cost estimates for the distributed application defined above must be provided. For each form of cost estimate, the distributed application input forms with which it can be legally associated are shown in parentheses.

- Expected computational workloads for each object and client. These workloads are estimates of the total execution time that will be required by each object and client, including CPU, communication and I/O time. The form and units of these estimates is TBD. (Application input form 1, 2 or 3).
- For each object, and for each operation defined on that object, the expected execution time for an arbitrary invocation of that operation, in units TBD. Also for each object, the estimated number of calls each object will make on each operation in every other object. (Application input form 2 or 3).
- A single input data set for the distributed application. While any data set is acceptable, the interpretation of the tool's output (see section 2) will be dependent upon this input data set. The extent to which the generated output allocation can be interpreted as an efficient allocation for the program given different input data sets is limited by the degree to which the input data set is representative (i.e., leads to a similar pattern of execution across objects and clients) of input data

sets anticipated for the application. (Application input form 3 only).

6.2.2.3. Processing environment

- For each of the maximum 400 sites (host computers) in the distributed processing environment, its relative speed expressed as a fraction of the fastest processor's speed (the relative speed for the fastest processor will be 1). No units are associated with this quantity since it is a ratio.
- A cost model reflecting the cost of data transmission over the communication medium, in units TBD. This model is expressed as a function of the host processor, the target processor, and the size of a message.
- A cost model reflecting communication system delays, units TBD. This model is expressed as a function of two parameters, the allocation of objects to nodes and the total message traffic for the application.

6.2.2.4. Objective function selection

One of the following objective functions is selected. Note that the application is expected to terminate as both quantities are finite.

- Application response time, in units TBD. This is the wall-clock execution time for the entire application, from its initiation to its termination.
- Total execution time for the application in units TBD. This is defined as the sum of the execution times of all application components, including components which execute in parallel (simultaneously).

6.2.2.5. Assignment constraints

Up to 1000 constraints may be placed on the assignment of objects and clients to processors. These may be either in terms of object assignments (assigning a particular object to a specified subset of nodes) or load constraints for processors (a particular processor must host no more than, or at least, a specified number of application components).

6.2.2.6. Search constraints on optimization

None, one, or both of the following constraints are provided for limiting the execution time for the allocation tool:

- Maximum time for search. This may be either in terms of the maximum number of solutions evaluated, the CPU time for the optimization, or the wall-clock time for the optimization.
- Termination criteria. This is expressed as an acceptable level of performance according to the objective function defined above.

6.2.3. Outputs

- An allocation of objects to processors. Qualitatively, this is to be as close to optimal as is feasible under the search termination constraints (see section 1.6). Given the nature of the problem, however, optimality cannot be guaranteed in a polynomial-time algorithm. Therefore, requirements for the output will be defined as follows: computation time for the optimization cannot be as large as an exponential function of the number of application components or processors (or any other input parameter); as noted above, the user may further constrain this time allotment. Within the time allotment, algorithms and heuristics will be employed for finding an allocation which yields high performance, and the allocation generated will be have the highest performance of all allocations that are evaluated during the optimization phase.

- The expected performance of the allocation defined in the previous bullet. This will be the value of objective function under that allocation. If termination criteria are specified as described under the second bullet in section 1.6, then either a) the expected execution time of the resulting allocation will not be greater than that quantity or b) explicit notification to the contrary will be provided if the desired level of performance is not achieved within the specified (as described in the previous bullet) time limits.

6.2.4. User interface

The tool provides an interactive interface. Where possible, inputs for the tool are taken from the project database (PDB) on the basis of user-supplied references to required PDB entities; the tool provides a tool-specific front end to the PDB. Inputs that, by necessity or choice, are not obtained from the PDB will be provided either directly to the tool or indirectly with the user providing references to data contained in files in the file system. Outputs from the tool may be saved in a file or in the PDB, at the user's discretion.

PART III

RELIABILITY ANALYSIS TOOL

1. Scope

1.1. Identification

This Software Top Level Design Document describes the top level design for the Computer Software Configuration Item (CSCI) identified as Reliability Analysis Tool, CSCI Number 3, of the Tool Integration Framework and Development Tools System. The requirements for the System appear in the Draft System/Segment Specification.

1.2. Overview

1.2.1. Purpose

The DISE environment supports the development of C^3 applications for execution in distributed processing environments. One of the most important characteristics of an effective C^3 application is survivability. Survivability is the ability to meet mission requirements in the event of hardware failures and can be measured in terms of reliability and availability. Successful development of *survivable* applications requires the ability to evaluate their reliability and availability characteristics.

Development decisions concerning reliability made at a later stage of development that lead to unacceptable levels of reliability can result in very costly redesigns. It is, therefore, important that developers have the capability for design-time analysis of reliability characteristics.

The Reliability Analysis Tool allows the developer to analyze the implications of processing environment (hardware) reliabilities on distributed software applications. It supports the developer in experimenting with alternative design decisions related to the application's decomposition, component functionality, processing environment and allocation of software components to computing platforms, each of which influences application reliability characteristics.

1.2.2. Role in the Integrating Framework

A second purpose for the Reliability Analysis Tool is to serve as a demonstration of the Tool Integration Framework, CSCI Number 1. The tool uses the Project Database (PDB) for the bulk of its I/O needs via the User Interface.

The tool provides a tool-specific front-end to the PDB. If desired, the inputs to the tool are taken directly from the PDB using user-supplied references to the required PDB entities. Alternatively, inputs to the tool may be read in interactively or from files in the local file system. The tool's output may be viewed and/or saved in the PDB or a file, at the user's discretion.

The architecture of the tool with respect to Cronus is shown in Figure 11. The Reliability Analysis Tool is a directly-coupled tool. It will interface to the PDB through the Tool Access Interface via Cronus.

1.2.3. Use and Functionality

The Reliability Analysis Tool accepts input data through a convenient user interface and generates output indicating the reliability. This is measured in terms of limiting availability (A), of the specified sub-system which is defined as

$$A = \frac{MTTF}{(MTTF + MTTR)}$$

where MTTF is the mean-time-to-failure and MTTR is the mean-time-to-repair of the component.

The tool takes information about the distributed application, the processing environment and the allocation of application components (UODs) to processors as input. Two aspects of the distributed

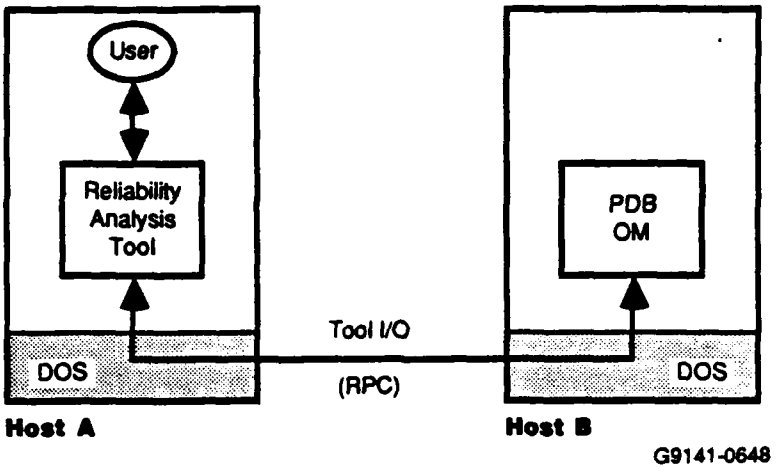


Figure 11. Reliability Analysis Tool and DISE

application are required: the names of the application components (client and object UODs) and call relationships between these components. In addition, the component of interest is named, which identifies a subsystem of the application components through a transitive closure on the call relationships. This subsystem can be viewed as a directed graph in which the edges specify dependencies. A set of annotations are input to the tool which specify probabilities and criticalities of these calls. The hardware characteristics of each component in the system, including the host computers (nodes) and communication links is given. An assignment of each application component (UOD) to one or more processors in the environment is given.

The tool generates the expected availability of the application subsystem *rooted* at the specified application component.

The DISE/Cronus hardware environment is, essentially, a network of SUN workstations on an Ethernet. These LANs may be connected over WANs in an Internet. The Ethernet environment can be modeled as individual processors networked via a shared medium. Logically, the system can be considered as a network of processors directly connected to each other, with all these communication links having identical failure probabilities.

We assume that processors fail independently and the physical environment is a shared medium. When a processing node is down, all the application components assigned to that node are unavailable. Hence, a limiting availability can be associated with the software components in the system.

The system is modeled as a directed graph. The nodes in the graph represent the UODs in the system and the edges represent call relationships between these UODs. These edges are annotated with call probabilities that reflect semantic dependencies between these UODs. For example, C_{ij} is the probability that UOD_i calls UOD_j during the execution of UOD_i. These probabilities specify the *dependency* of one UOD on another.

A *transitive* call probability of UOD_i calling UOD_j in the selected sub-system can be computed. This probability expresses the *dependency* of UOD_i on any UOD_j in the system. All possible traversal paths from i to j and the annotations on these edges have to be considered in computing this transitive call probability.

A matrix with these dependencies can be constructed. For the given UOD_I, a vector D_I is computed for all specifying these transitive call probabilities from UOD_I to all the other UODs in the specified sub-system. The call graphs are assumed to be acyclic. An embellishment to the algorithm to handle cyclic call dependencies will be discussed later in this section.

Having computed this dependency matrix, the call graph constructed for the application can be mapped onto the processing environment. The given allocation specifies the assignment of each UOD to one of the processors in the system. A *need* for (or the reliance on) the different processors in the system/sub-system can be derived from the dependency matrix knowing the assignment of UODs to processors.

A vector of these *reliance* or *need* values can be computed to denote the system/sub-system's need for each of these processors. This reliance vector will reflect the application sub-system's dependence on the hardware resources in the processing environment. Given the failure probabilities of these processors (P_i), the availability of the system can be computed. We now have the failure probabilities of the processors and a measure of the degree of dependency of the system on each of these processors.

The different failure states of the system are determined by considering all combinations of failure states of the processors in the system. The probability of the processing environment being in each of these states weighted by the corresponding reliance factor yields each state's affect on application availability.

1.2.4. Development Phases

Development of the Reliability Analysis Tool will take place in four phases:

Phase 1:

Develop a convenient user interface. The user interface will offer window-based data-entry facilities. In addition, it will provide a tool-specific interface to the Project Database for retrieval and storage of tool data at the user's discretion. Completing this phase will allow the tool to be integrated with the PDB when the PDB becomes operational, even though the tool will exhibit minimal functionality.

Phase 2:

We assume that processors fail independently and the physical environment is a shared medium. This is the minimum acceptable functionality of the tool required to be able to demonstrate the tool's utility and the tool integration framework.

Phase 3:

Replication, as implemented in Cronus will be considered (given read/write quorums and number of copies).

Phase 4:

The shared medium networks - like Ethernet LANs - are interconnected in an Internet environment. These interconnected Ethernet clusters can be modeled as a network of computing platforms with their own hardware reliability characteristics. In Phase 3, a hierarchical model is used. The availability of the subsystems on the Ethernets is computed as before. These availabilities and the communication link - between these LANs- availabilities are considered, and the availability of the overall Internet system is computed. Alternate paths between LANs may be determined by routing mechanisms or complete connectivity may be assumed.

Given the existence of routing and communication links, the reliability problem is to compute the proability that the given *component-of-interest* that identifies the system/subsystem can reach every other component in the system. Graph theoretic algorithms to compute Source to Multiple

Terminal (SMT) reliabilities in such an environment for certain special cases (network topologies) are available in the literature. They consider independent and dependent processor and link failure probabilities, dynamic routing mechanisms, and special network topologies, but none of them consider call probabilities or call criticalities.

Methods of extending the tool's functionality by implementing or adapting one or more of these algorithms to meet our requirements will be identified. A hierarchical reliability computation model will be implemented to allow reliabilities of interconnected networks of different topologies to be estimated.

- Identify relevant algorithms in literature;
- Identify methods of extending tool's functionality by incorporating these algorithms;
- Implement a hierarchical reliability computation model:
    - different connectivity and routing assumptions;
    - different network topologies;
    - link failure probabilities;

Phases 3 and 4 will be implemented only if sufficient time and effort are available.

1.3. Introduction

This document presents a top level functional and abstract data type decomposition of the Reliability Analysis Tool CSCI into Top-level Computer Software Components (TLCSCs) and the interfaces among TLCSCs of this CSCI. It also defines the interfaces between this CSCI and other elements of the Tool Integration Framework and Development Tools System. Included in the Notes (section 6) are an external interface definition and a detailed requirements specification derived from the System/Segment Specification.

2. Referenced Documents

- *Draft System/Segment Specification*, DOS Design Application Tools, Contract No. F30602-87-C-0104, CDRL No. 3, 25 August, 1988, RADC/COTD, (Honeywell Inc., Corporate System Development Division).

3. Requirements

3.1. CSCI Architecture

Figure 12 shows the top level functional decomposition for this Reliability Analysis Tool CSCI. The elements of the decomposition, each a TLCSC, are functions and abstract data types (ADTs). The primary control loop is captured in the function Main.

3.2. Functional Allocation

The requirements for this CSCI are defined in the System/Segment Specification. In particular, the requirements for the Reliability Analysis Tool specified in section 3.1.6.3 of the System/Segment Specification identify requirements for four functions: Static Analysis, Subsystem Resolution, Reliability Computation and User Interface. Identified below are the TLCSCs to which each of the four functions has been allocated:

- Static Analysis: This is implemented by function TLCSCs 3.6.1.2 (Initialize) and ADT TLCSC 3.6.2.1 (Applic).
- Subsystem Resolution: This is implemented by function TLCSCs 3.6.1.4 (Process_command) and ADT TLCSC 3.6.2.1 (Applic).

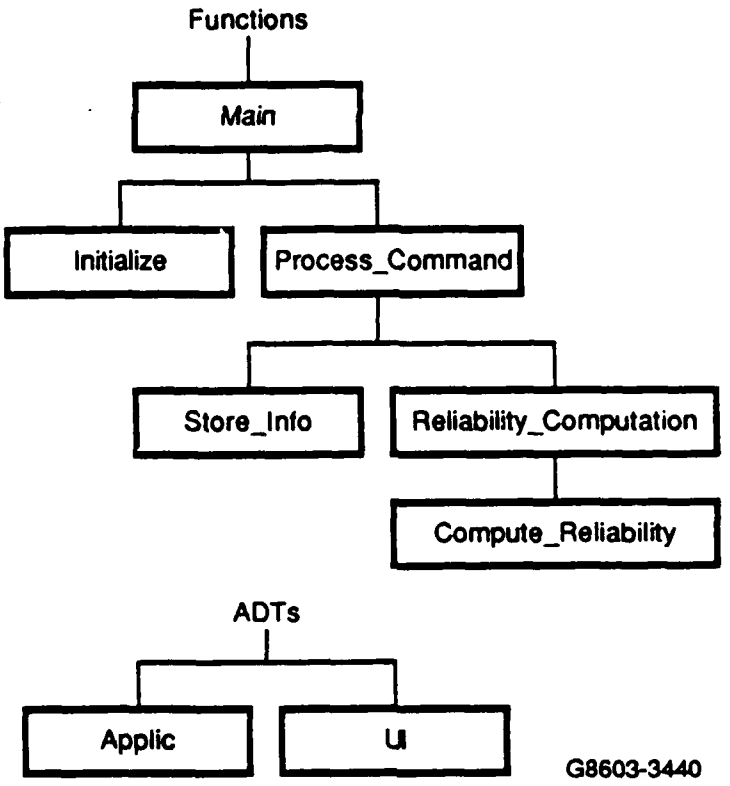


Figure 12. Reliability Analysis Tool Functional Decomposition

- Reliability Computation: This is implemented by function TLCSCs 3.6.1.5 (Compute_reliability), 3.6.1.6 (Reliability_computation) and ADT TLCSC 3.6.2.1 (Applic).
- User_Interface: This is implemented by function TLCSCs 3.6.1.1 (Main), 3.6.1.2 (Initialize), 3.6.1.3 (Store_info) and ADT TLCSC 3.6.2.2 (User_Interface).

3.3. Memory and Processing Time Allocation

Not specified.

3.4. Functional Data Flow

The functional data flow for the Reliability Analysis Tool is shown in Figure 13. Boxes in the diagram represent individual function and ADT TLCSC's, the latter occupying the bold-lined boxes. Arrows between boxes indicate data flow, and have been annotated to show the primary data that is passed from one TLCSC to another.

3.5. Global Data

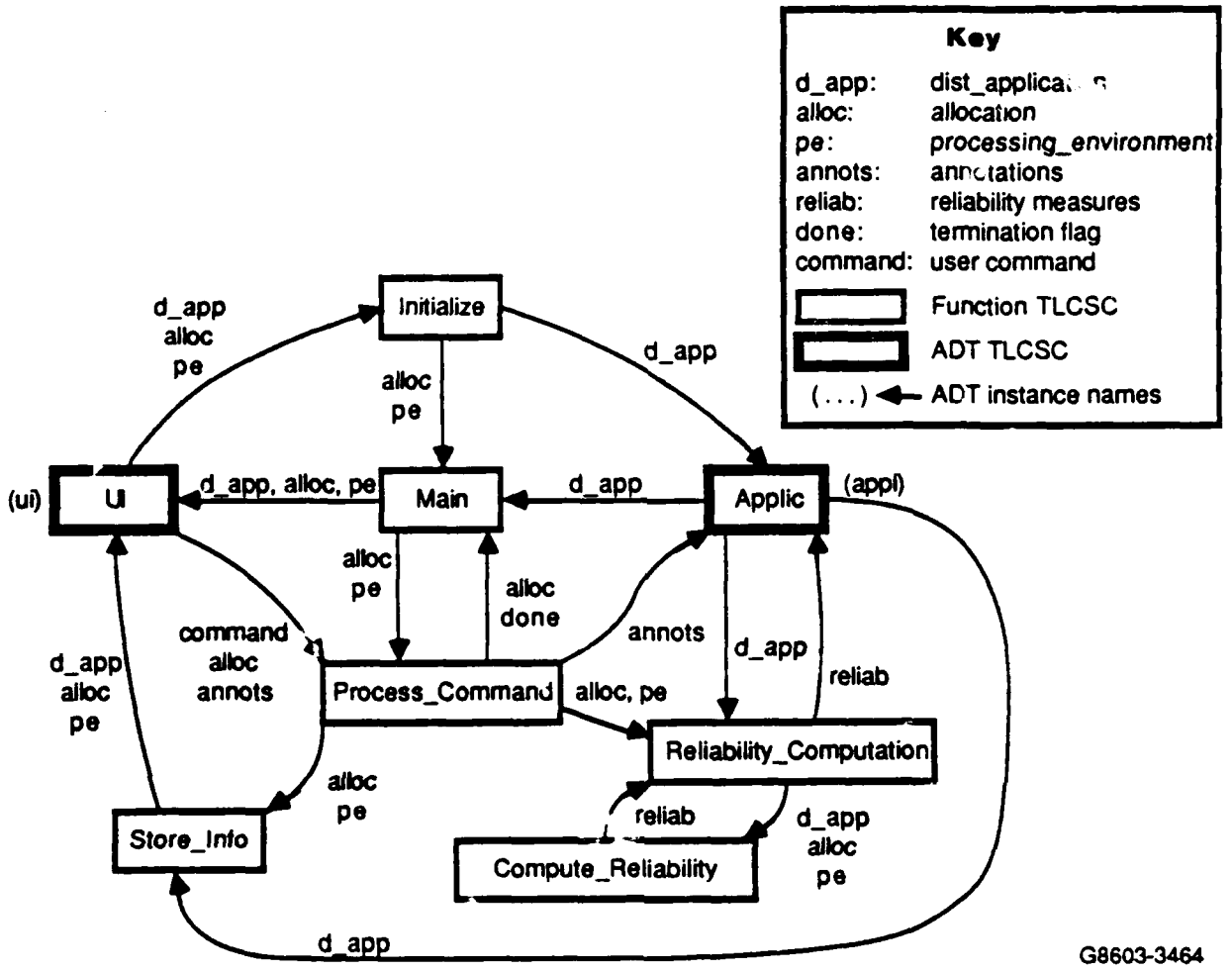


Figure 13. Reliability Analysis Tool Data Flow

3.5.1. Global Data Types

```
Comp_Type = (Client, Object);   (* Type of component *)

Comp_Name = STRING;             (* Name of client or object *)

Proc_Name = STRING;             (* Name of processor/node *)

Comp_Names = SET OF Comp_Name;
            (*  Names of the objects/clients comprising an application. *)

annot = RECORD
    weight : REAL;
            (* The edge weight indicates the probability of calls from
            one application component to another for an arbitrary invocation
```

```
            of the calling component. *)
    criticality : REAL
            (* This specifies the criticality/necessity of this call *)
END;

Call_Info = RECORD (* info about calls for a particular UOD *)
    name : Comp_Name;
    annots : ARRAY[Comp_Names] of annot;
END;

Call_Set = SET OF Call_Info;

Reliability : REAL;

Comp_Info = RECORD
    name : Comp_Name;
    kind : Comp_Type;
    ext_spec_ref,
    ext_body_ref : External_File;
    reliability : Reliability;
END;

Comp_Set = SET OF Comp_Info;

Processor = RECORD        (* Information about a processor *)
    proc_name : Proc_Name;
    proc_reliability : Reliability;
END;

Proc_Environ = RECORD        (* Information about the processing environment *)
    processors : SET OF Processor;
            (* set of processors comprising the processing environment *)
    comm_reliability : MATRIX [Proc_Name, Proc_Name] OF Reliability;
            (* reliability of the communication medium *)
END;

MAPPING FROM SET X TO SET Y = (* Total function from X to Y.
    Abstract model:    a set of pairs <x,y>.
    Operations:    For x in X and y in Y:
                   assign(x, y) : associates y with x.
                   lookup(x) returns y : returns associated y for x *)

Allocation = MAPPING FROM Comp_Names TO SET OF Proc_Name;
    (*  The allocation of objects/clients to processors in the system *)

Command = (def_alloc, def_annotations, focus, compute, save, quit, reset);
    (* User commands *)
```

3.5.2. Global Variables

```
(*** INPUT VARIABLES ***)
names : Comp_Names;
        (* names of clients/objects in the application *)
calls : Call_Set;
        (* calls (probabilities & criticalities) for the entire application *)
proc_env : Proc_Environ;
        (* processing environment: nodes and comm links *)
alloc : Allocation;
        (* given allocation for this reliability estimation *)
head : Comp_Info;
        (* handle for entire application *)

(*** ADT INSTANCES ***)
ui : UI;
        (* the user interface ADT instance *)
appl : Applic;
        (* the Application ADT instance *)

(*** GLOBAL DATA STRUCTURES ***)
done : BOOLEAN;
        (* termination flag : to exit from tool *)

(*** OUTPUT VARIABLES ***)
reliab : Reliability;
        (* the reliability estimate *)
```

3.6. Top Level Design

3.6.1. Function TLCSCs

3.6.1.1. Main TLCSC

3.6.1.1.1. Identification

This is the main function in the CSCI. It encapsulates the primary control loop for the tool. Following initialization, it iteratively processes a user command and provides a visual feedback of the changes made by displaying the current state of the application.

3.6.1.1.2. Inputs

There are no inputs to this function. In the case of this function, inputs would be command-line arguments provided at the time of tool invocation.

3.6.1.1.3. Local Data

Local data for the main program is global data for the tool, and global data is identified above in section 3.5.2.

3.6.1.1.4. Processing

```
BEGIN Main;

    Initialize(ui, appl, head, alloc, proc_env);
    LOOP
        ui := UI$display(ui, Applic$gen_comps(appl), Applic$gen_calls(appl), alloc, proc_env);
        Process_command(ui, appl, head, alloc, proc_env, done);
    UNTIL done;

END Main.
```

3.6.1.1.5. Outputs

No outputs are generated by this function within the CSCI. The external interface generated, in essence, by this function is described in section 6.1.

3.6.1.2. Initialize TLCSC

3.6.1.2.1. Identification

This function initializes the User Interface and obtains, from the user, necessary input parameters. These parameters may come from the Project Database (PDB). The application model (Applic ADT instance) is created, using the component names and calls for the entire application.

3.6.1.2.2. Inputs

None.

3.6.1.2.3. Local Data

```
comps : Comp_Names; (* just the NAMES of components *)
calls : Call_Set;    (* calls between the components *)
```

3.6.1.2.4. Processing

```
BEGIN Initialize;

    ui := UI$init();
        (* initialize the User_Interface ADT *)
    <names, calls, proc_env, alloc> := UI$get_inputs(ui);
        (* get desired input parameters => from the PDB or directly from the user *)
    appl := Applic$create(comps, calls);
        (* create an instance of the Applic ADT to model the application *)
    head := UI$get_coi(ui);
        (* get place holder for entire application *)

END Initialize;
```

3.6.1.2.5. Outputs

```
ui : UI;
appl : Applic;
head : Comp_Info;
```

alloc : Allocation;
proc_env : Proc_environ;

3.6.1.3. Store_info TLCSC

3.6.1.3.1. Identification

This function stores the current application information in the PDB. This information includes component reliabilities, call annotations, component allocation, and processing environment attributes.

3.6.1.3.2. Inputs

ui : UI;
appl : Applic;
head : Comp_Info;
alloc : Allocation;
proc_env : Proc_environ;

3.6.1.3.3. Local Data

None.

3.6.1.3.4. Processing

```
BEGIN Store_info;

    tmp_appl = appl;
        (* store current sub-system *)
    appl := Applic$focus(appl, head);
        (* get the entire application *)
    ui := UI$save(ui, Applic$gen_comps(appl), Applic$gen_calls(appl), alloc, proc_env);
        (* store relevant information in the PDB *)
    appl = tmp_appl;
        (* reset current sub-system *)

END Store_info;
```

3.6.1.3.5. Outputs

None.

3.6.1.4. Process_command TLCSC

3.6.1.4.1. Identification

This function processes a single user command at a time. These commands allow the user to manipulate system parameters (such as the allocations and call annotations and processor reliabilities), focus on a particular set of components in the application, save the current parameters, and perform reliability computations.

3.6.1.4.2. Inputs

ui : UI;
appl : Applic;

```
head : Comp_Info;
alloc : Allocation;
proc_env : Proc_environ
```

3.6.1.4.3. Local Data

None.

3.6.1.4.4. Processing

```
BEGIN Process_command;

    CASE UI$command(ui) OF
        def_appl :          (* define a new application *)
            {
                <comps, calls, proc_env, alloc>  := UI$get_inputs(ui);
                appl := Applic$create(comps, calls);
                head := Applic$gen_coi(Applic);
            }
        def_annotations :   (* define/modify annotations on the call graph *)
            appl := Applic$annotate(appl, UI$get_annots(ui));
        def_alloc :         (* define/modify allocation *)
            alloc := UI$get_alloc(ui);
        def_environ :       (* define/modify the processing environment *)
            proc_env := UI$get_environ(ui);
        compute :           (* compute the reliability of appl *)
            Reliability_computation(appl, alloc, proc_env);
        focus :             (* on application component named by user *)
            appl := Applic$focus(appl, UI$get_coi(ui));
        reset :             (* reset appl to include the entire application *)
            appl := Applic$focus(appl, head);
        save :              (* save the current status of appl *)
            Store_info(ui, appl, head, alloc, proc_env);
        quit :              (* exit loop *)
            done := TRUE;
    END;

END Process_command;
```

3.6.1.4.5. Outputs

```
ui : UI;
appl : Applic;
alloc : Allocation;
done : BOOLEAN;
```

3.6.1.5. Compute_reliability TLCSC

3.6.1.5.1. Identification

This function implements the reliability analysis algorithm. It performs the actual reliability computation for the application component under study in the context of the current allocation and processing

environment.

3.6.1.5.2. Inputs

component_of_interest : Comp_Info;
comps_info : Comp_Set;
calls : Call_Set;
alloc : Allocation;
proc_env : Proc_Environ;

3.6.1.5.3. Local Data

None.

3.6.1.5.4. Processing

BEGIN Compute_reliability;

(* Perform reliability analysis on the application. The return value will be the expected limiting availability for the specified component of interest (for example, a particular client). It will be a function of the call relationships which involve that component (directly or indirectly), the assignment of each of these components to processors, and reliability characteristics of the processing environment.

We assume that processors fail independently and the physical environment is a shared medium. When a processing node is down, all the application components assigned to that node are unavailable. Hence, a limiting availability can be associated with the software components in the system. For convenience, the following variables are defined:

$$p_i = \textit{probability that Unit-Of\_Distribution}_i\ (UOD_i)\ \textit{is available}$$

$$P_i = \textit{probability that processor } i \textit{ is available}$$

$$A_i = \textit{availability of sub-system rooted at } UOD_i$$

$$\Gamma_I = \textit{set of processors in the system that host the UODs in the sub-system rooted at } I$$

$$\Omega_I = \textit{set of application components in the sub-system rooted at } I\ (\textit{comps\_info} : UODs)$$

$$a : \textit{Allocation}\ (\textit{a mapping of } UODs \rightarrow \textit{processors})$$

$$UNAVAIL(i) = \textit{unavailability of sub-system rooted at } UOD_i$$

$$C_{ij} = \textit{probability of } UOD_i \textit{ calling } UOD_j$$

$$B_{ij} = \begin{cases} 0 & UOD_i \text{ and } UOD_j \textit{ are co-assigned} \\ 1 & \textit{otherwise} \end{cases}$$

I = *the component-of-interest*

The term $R_i(a)$, expresses the reliability for the system/sub-system rooted at UOD_i as a function of a given assignment a. This tool computes R_i for the given assignment a.

The system is modeled as a directed graph. The nodes in the graph represent the UODs in the system and the edges represent call relationships between these UODs. These edges are annotated with call probabilities that reflect semantic dependencies between these UODs. For example, C_{ij} is the probability that UOD_i calls UOD_j during the execution of UOD_i. These probabilities specify the *dependency* of one UOD on another.

Initially, if the shared medium is assumed to be fully reliable and all the call probabilities are 1, the sub-system rooted at UOD_I will be unavailable if any UOD it calls directly or indirectly is unavailable. The reliability of the sub-system rooted at UOD_I as measured by its expected availability is given by:

$$A_I = \prod_{i=1}^{|\Gamma_I|} P_i \qquad \text{(System Reliability)}$$

i.e., the system is unavailable if any of the processors in the system fail. Still keeping the call probabilities at 1, if the failure probability of the shared medium is considered, the availability of the system is computed by

$$A_I = P_m \times \prod_{i=1}^{|\Gamma_I|} P_i \qquad \text{(System Reliability)}$$

where P_m is the failure probability of the shared medium. As every remote call between 2 UODs has to go via the shared medium, its failure always results in the failure of system (or sub-system).

Now if the call probabilities are less than 1, the effect of the called UOD's unavailability on the caller's availability is reduced. A *transitive* call probability of UOD_i calling UOD_j in the selected sub-system can be computed. This probability expresses the *dependency* of UOD_i on any UOD_j in the system. All possible traversal paths from i to j and the annotations - probabilities - on these edges have to be considered in computing this transitive call probability. Let

$$D_{ij} = \textit{dependency of } UOD_i \textit{ on } UOD_j$$

A matrix with these dependencies can be constructed. For the given UOD_I, a vector D_I is computed for all $1 \le j \le |\Omega_I|$ and $j \ne I$. At this stage, the call graphs are assumed to be acyclic. An embellishment to the algorithm to handle cyclic call dependencies will be discussed later in this section.

compute_dependency_matrix(I, D) => D_I vector
{
    begin
    traverse the call graph and number the UODs;
    for all j such that $j \in \Omega_I$ **do**
        if j = I **then return** $D_{Ij} = 1$; **continue**;
        compute PATH = the set of paths from UOD_I to UOD_j;
        let $k \in PATH$;
        let $l_k = \prod$(*all call probabilities in path k*);

let D_{Ij} = P(UOD_j being called directly or indirectly by UOD_I);

compute $D_{Ij} = 1 - \prod_{k=1}^{|PATH|} (1 - l_k)$;

end

return;

}

The call graph constructed for the application can be mapped onto the processing environment. The allocation specifies the assignment of each UOD to one of the processors in the system. Hence, the dependency on each of the UODs in the system can be translated to a *need* for the different processors in the system/sub-system. Let

$$N_j = \textit{reliance on processor } j$$

An array of these *reliance* or *need* values can be computed to denote the system/sub-system's need for each of these processors. If UOD_I is the component-of-interest:

compute_need_matrix(I, N) => N_j matrix

{

begin

let a(I) = p_I;

for all j such that $j \in \Gamma_I$ do

if j = p_I then $N_j = 1$; continue;

let CO_ASSIGN(j) = set of UODs assigned to processor j;

let N_j = reliance on processor j;

$N_j = 1 - \prod_{k=1}^{|CO\_ASSIGN(j)|} (1 - D_{Ik})$;

end

return;

}

The reliance matrix will reflect the application sub-system's dependence on the hardware resources in the processing environment. Given the failure probabilities of these processors (P_i), the availability of the system can be computed. We now have the failure probabilities of the processors and a measure of the degree of dependency of the system on each of these processors.

compute_avail() => *availability* A_I

{

begin

let Θ be the states of the system (combination of processor failure states);

for each j such that $j \in \Theta$ do

let F(j) be the probability of the system being in state j;

let *UP* be the set of processors that are UP;

let *DOWN* be the set of processors that are DOWN;

let $\bar{P}_j$ be the failure probability of processor j = 1 - P_j;

$$F(j) = \left[\left[\prod_{k=1}^{|UP|} P_j\right] \times \left[\prod_{k=1}^{|DOWN|} \bar{P}_j\right]\right];$$

let PN(j) be the cumulative reliance on the *unavailable* processors in system state j;

compute PN(j) = $1 - \prod_{k=1}^{|DOWN|} \left(1 - N_k\right)$;

end;

let U_I = sub-system *unavailability*;

$$U_I = \sum_{j=1}^{|\Theta|} \left(F(j) \times PN(j) \right);$$

$A_I = 1 - U_I$;

end

return

}

The different states of the system are determined by considering the combination of the possible states of the processors in the system. The probability of the processing environment being in each of these states and its effect on application availability is computed. These system states are weighed by the corresponding reliance (need) probabilities.

The above algorithm estimates the availability of the system conservatively. While not exact, the algorithm errs by always *under estimating* the availability of the system. The following embellishments to the algorithm result in a more accurate estimate.

Let UOD_i, UOD_j and UOD_k be three UODs in the system and let $A_{jk} = 1$, i.e., UOD_j and UOD_k are assigned to the same processor, say n. Moreover, let a path from UOD_i to UOD_j include UOD_k. If processor n is unavailable, UOD_k is inaccessible. UOD_j is, therefore, inaccessible. Hence, while computing the need for processor n, (N_n), if we consider the contributions of paths D_{ik} *and* D_{ij}, we will over estimate the reliance of the system on processor n. Hence, when computing the transitive call probability D_{ij}, contributions made to it by paths going through UOD_k are ignored. This is to ensure that N_n is not over estimated.

This algorithm correctly estimates the effect of single failures on the system's availability. While the probability of concurrent multiple failures is very small (given reliable processors and independent failure rates), their effect is more accurately estimated by applying corrections to the algorithm in certain special cases.

For example, let a system consist of three UODs, UOD_i, UOD_j and UOD_k. Let them be assigned to three processors a, b, and c respectively. Moreover, let UOD_j be on every path from UOD_i to UOD_k. Let UOD_i be the component of interest. In computing system availability, if processor b is down, UOD_k on processor c is unreachable. Hence, given processor b is down, the availability of the system is unaffected by the state of processor c. To account for this, a compensating correction has to be applied.

To accommodate cyclic calling graphs, the effect of these recursive call relationships on the dependency values D_{ij} has to be correctly determined. With the edges in the graph representing call relationships, a reachability matrix can be computed to determine the transitive call dependencies. From this, the cycles in the graph can be detected and treated properly.

END Compute_reliability;

3.6.1.5.5. Outputs

reliab : Reliability; (* estimated reliability *)

3.6.1.6. Reliability_computation TLCSC

3.6.1.6.1. Identification

This function initiates the reliability analysis on the appropriate application information in the context of the current allocation and the processing environment. It then updates the application information with the reliability measure.

3.6.1.6.2. Inputs

appl : Applic;
alloc : Allocation;
proc_env : Proc_Environ;

3.6.1.6.3. Local Data

reliab : Reliability;

3.6.1.6.4. Processing

```
BEGIN Reliability_computation;

    reliab := Compute_reliability(Applic$gen_coi(appl), Applic$gen_comps(appl),
                Applic$gen_calls(appl), alloc, proc_env);
      (* compute the reliability of the sub-system rooted at coi *)
    appl := Applic$update(appl, reliab);
      (* update the appropriate Applic ADT instance with this reliability measure *)

END Reliability_computation;
```

3.6.1.6.5. Outputs

appl : Applic;

3.6.2. ADT TLCSCs

3.6.2.1. Applic TLCSC

3.6.2.1.1. Identification

This ADT models the application. In particular, information about components and their relationships is maintained in a form suitable for the graph-type operations to be performed.

3.6.2.1.2. Abstract Model

```
component_of_interest : Comp_Info;
            (* root of the identified subsystem *)
applic_components : Comp_Set;
            (* application components: nodes of the call graph *)
invocations : Call_Set;
            (* call relationships : edges of the call graph  *)
```

3.6.2.1.3. Operations

create: Comp_Names, Call_Set --> Applic;

(* Creates an App_Model. *)

focus: Applic, Comp_Name --> Applic;

(* Establish Comp_Name as the current component of interest in the Applic. Component sets and call sets generated from gen_calls and gen_comps, below, will be with respect to this component, i.e. the subgraph with Comp_Name as the root. The set of components, then, will be those components called directly or indirectly by Comp_Name, and likewise with the call set. *)

annotate: Applic, Call_Set --> Applic;

(* Weights in the given Call_Set are incorporated into the Applic. *)

gen_coi: Applic --> Comp_Info;

(* Returns the current component of interest. *)

gen_comps: Applic --> Comp_Set;

(* Generates the set of components called directly or indirectly by the component of interest. *)

gen_calls: Applic --> Call_Set;

(* The set of calls in the Applic, relevant to the current component of interest, is returned. *)

update: Applic, Reliability --> Applic;

(* Associate the Reliability with the current component of interest in Applic. *)

3.6.2.2. UI TLCSC

3.6.2.2.1. Identification

This ADT encapsulates the user interface. It manages the display device and input devices. It also encapsulates the interface with the PDB in support of tool I/O needs as dictated by the user.

3.6.2.2.2. Abstract Model

screen : User_Interface_Display;

3.6.2.2.3. Operations

init: --> UI;

(* Create a user interface. *)

get_inputs: UI --> <Comp_Names, Call_Set, Proc_Environ, Allocation>;

(* Obtain 4 input parameters from the user, through the PDB if specified by the user. *)

get_annots: UI --> Call_Set;

(* Get a set of annotations from the user. *)

get_alloc: UI --> Allocation;

(* Get an allocation from the user. *)

get_environ: UI --> Proc_Environ;

(* Get the processing environment *)

get_coi: UI --> Comp_Info;

(* Get the current component of interest (coi) for the user. *)

display: UI, Comp_Set, Call_Set, Allocation, Proc_Environ --> UI;

(* Display the Applic to the user showing components, calls, annotations, reliabilities, the allocation and the processing environment. Refresh the screen to reflect changes in appl's state. *)

command: UI --> Command;

(* Return a user command. *)

save: UI, Comp_Set, Call_Set, Allocation --> UI;

(* Save the current application information in the PDB. This includes the application components, calls with annotations, component reliabilities, the allocation and the processing environment. *)

3.7. Adaptation Data

This tool executes on a Sun workstation and uses the Sunview windowing system for the user interface. Since the tool is developed as a Cronus application, the host workstation environment must run the Cronus distributed operating system.

6. Notes

6.1. Interfaces

As an application development tool, the Reliability Analysis Tool interfaces with the following external entities:

- Cronus distributed operating system: The tool, like any Cronus application, may require access to various Cronus system object managers.
- Project Database: The PDB is a specific Cronus object manager to which the Reliability Analysis Tool interfaces for various I/O needs. The data items potentially obtained from the PDB include information about the components in the distributed application, target processing environment reliability attributes and the allocation of application components to processors. The reliability measures computed for various application components along with possible modifications to the allocation of application components to processors and information about the components in the distributed application may all be stored in the PDB. The interface itself is defined by the operations exported by the PDB object.

6.2. Requirements Specification

This subparagraph provides a condensed version of the requirements specified for this CSCI. These requirements are derived directly from the System/Segment Specification identified in paragraph 2.

6.2.1. Purpose

The Reliability Analysis tool accepts input data through a convenient user interface and generates outputs indicating the reliability, measured in terms of limiting availability, of distributed application components.

6.2.2. Inputs

6.2.2.1. Distributed application

A distributed application is composed of one or more client modules and the set of objects to which the clients pass messages, directly or indirectly. Two aspects of the distributed application are required input:

- The names of these application components (clients and objects).
- Call relationships between the components. For each component, there is a list of the objects called (potentially) by that component.

6.2.2.2. Hardware Reliability Characteristics

The hardware environment consist of up to 400 host computers and communication subsystems connecting the host computers. Each host computer is identified by a unique name; each communication subsystem is identified by the set of host computers it connects.

The reliability characteristics of each hardware component, including host computers and communication systems, is provided. Within the context of this tool, reliability is measured in terms of the limiting availability of components, and it is assumed that components fail independently. The limiting or "steady-state" availability, A, is defined as:

$$A = MTTF / (MTTF + MTTR)$$

where MTTF is the mean-time-to-failure and MTTR is the mean-time-to-repair of the component. Note that this quantity (which is in the range [0..1] and is unit-less) is independent of the nature of the underlying distributions of failure and repair times.

The limiting availability is specified for each host computer in the environment and for each communication subsystem. .sh 4 "Allocation of Application Components"

For each application component identified in section 1.1. a list of host computers is given indicating the host computers upon which that component will execute. Each component's list contains the name of at least one host computer, and contains more than one if and only if it is replicated.

6.2.2.3. Subsystem Specification

A designation of the application subsystem for which reliability analysis will be performed is provided by naming the desired application component (object or client). Naming an application component defines a subsystem of application components through the transitive closure on the call relationships specified in section 1.1 for that component. For example, naming an object which calls two other objects, neither of which calls any other objects, defines a subsystem consisting of those three objects.

6.2.2.4. Subsystem Annotations

A subsystem, once defined, can be viewed as a directed graph. The edges in the graph indicate that one application component may call the other. This means that the first object depends on the second object in the sense that availability of the first component depends on the availability of the second component. However, a variety of runtime factors influence the actual degree to which one component's availability is influenced by another.

- An object may continue to execute successfully after realizing that a called object is not available. This eliminates the called object's effect on the calling object's availability.
- An object may not always call another. During execution, there may be a low probability that one object actually calls another. Therefore the negative effect of the called object's availability on the caller will be reduced.

Each edge in the subsystem graph is annotated with a number in the range [0..1] reflecting the above two factors. For example, a 1 indicates that the called object is necessary and that it is called every time. A 0 indicates that the calling object can continue executing despite failure of the called object's host. A value, say, of 0.5 indicates that the calling object calls the other object 50% of the time that the caller is invoked.

6.2.3. Output

The tool generates the expected availability of the application subsystem. This quantity is based on the reliability characteristics of the hardware environment, the placement of application components within that environment, and the patterns of interactions between the components of the subsystem.

6.2.4. User interface

The tool provides an interactive interface. Where possible, inputs for the tool are taken from the project database (PDB) on the basis of user-supplied references to required PDB entities; the tool provides a tool-specific front end to the PDB. Inputs which by necessity or choice are not obtained from the PDB are provided either directly to the tool or indirectly with the user providing references to data contained in files in the file system. Inputs not from the PDB are checked for consistency. Outputs from the tool may be saved in a file or in the PDB, at the user's discretion.

MISSION
of
Rome Air Development Center

RADC plans and executes research, development, test and selected acquisition programs in support of Command, Control, Communications and Intelligence (C^3I) activities. Technical and engineering support within areas of competence is provided to ESD Program Offices (POs) and other ESD elements to perform effective acquisition of C^3I systems. The areas of technical competence include communications, command and control, battle management information processing, surveillance sensors, intelligence data collection and handling, solid state sciences, electromagnetics, and propagation, and electronic reliability/maintainability and compatibility.